VLSI Concurrent Computation for Music Synthesis

John Wawrzynek

Computer Science Department
California Institute of Technology

5247:TR:87

VLSI Concurrent Computation for Music Synthesis

Thesis by
John Wawrzynek

In Partial Fulfillment of the Requirements
for the Degree of
Doctor of Philosophy

California Institute of Technology
Pasadena, California

1987

(Submitted April 1, 1987)

5247:TR:87

Acknowledgments

I would like to express my sincere thanks to Carver Mead. He has been my advisor, friend, and teacher throughout my stay at Caltech. Carver has been the origin, either directly or indirectly, of most of the ideas in this thesis. He has shown me and many others the way to perform research in computer science and is a great inspiration.

Several other people have been responsible for and have worked on the music project at Caltech. Lounette Dyer developed the software interface to our prototype system and did much of the work on developing parameters for the musical instrument models. John Pierce and Max Mathews jointly stirred up interest in computer music at Caltech, and for that I am very grateful. They have been very supportive through the years. Tzu-Mu Lin worked as my partner on the *n*MOS design of the chip. Hsui-Lin Liu did work on physical modeling of musical instruments. Many thanks to Charles Smith and the System Development Foundation for funding this research and computer music research at other institutions.

Special thanks are due Telle Whitney for her constant support, her critiques and numerous discussions.

Dick Lyon taught me much about integrated circuit design and signal processing and provided many detailed critiques of my work and drafts of this thesis. Ron Ayres wrote the ICL language that we used to compile the *n*MOS version of the chip. Later versions were assembled using tools developed by Massimo Sivilotti and Glen Gribble. Thanks to George Lewicki and the entire MOSIS crew for managing fabrication of the chips. Sandy Frey helped to organize my work into a thesis outline. Thanks to Calvin Jackson for teaching me about typography and helping to format this thesis. Many thanks also to all the members of our research group, they have all helped to make my stay at Caltech a pleasant and enlightening experience.

Finally, I thank my Thesis Committee: Al Barr, Richard Feynman, Dick Lyon, Carver Mead, and Chuck Seitz.

Abstract

This thesis presents a very large-scale integrated circuit (VLSI) approach to the generation of musical sounds. The approach allows the generation of rich musical sounds using models that are easy to control and have parameters corresponding to many of the physical attributes of musical instruments. The generality of the approach for music synthesis is demonstrated by presenting several primitive sound generation mechanisms. Utilizing these primitives, several musical instruments are assembled to produce struck, plucked, and blown sounds. Refinements of the instruments are easily accomplished by adjusting or rearranging different functional components. A concurrent computing engine supporting the sound generation mechanisms is presented along with details of its VLSI implementation. Involved in the implementation is a new CMOS design methodology. Several alternative architectures for the computing engine are also presented and studied.

Table of Contents

List of Figures and Tables

Introduction

Very large-scale integrated circuit (VLSI) technology is here and is changing the way scientists work. It is a powerful medium that is flexible enough to support a wide variety of ways of computing. With the help of VLSI we have an opportunity to rethink old problems and tackle new ones. Ideas that were once only reasoned about and simulated can now be built and experimented with.

Often in computer science, researchers have designed and built machines based on the capabilities of the technology rather than on the needs and inherent properties of particular applications, resulting in machines with impressive raw computing power, but not efficient at any one problem. An alternative approach is to design computers *specifically* for a particular class of problems. VLSI and quick turnaround prototyping give us the opportunity to pursue such a tactic.

Some researchers in computer science study problems only at a theoretical level and at best simulate their solutions. This approach has the drawback that invalid assumptions may remain unchecked and plague the work forever. In this thesis I present a solution to a problem by bringing together theory and implementation in VLSI.

The generation of realistic musical sounds is an interesting problem for several reasons. It has not been studied with respect to VLSI, although VLSI holds the possibility for large benefits. Sound synthesis is representative of many other computational tasks—therefore, any insight and understanding gained is likely to be applicable in other areas. Besides being computationally expensive, musical sound synthesis faces directly into the issues of machine-man interface.

Sound synthesis is not an easy problem; partly because human hearing and perception are not well understood. But listening tests can be performed, making the problem more tractable. It is clear when one is progressing in the right direction. This is an attribute sound synthesis shares with computer graphics—results are clearly visible. Also, in the same way that computer graphics can help us understand human vision, computer sound synthesis can help us understand human hearing.

Sounds that come from physical sources can be naturally represented by differential equations in time. Since there is a straightforward correspondence between

differential equations in time and finite difference equations, we can model musical instruments as simultaneous finite difference equations. Musical sounds can be produced by solving the difference equations that model instruments in real time.

The computational bandwidth that is needed to compute musical sounds is enormous. For the sampled waveform representation of sound, we need to produce samples at a rate of about 50 K samples/sec. If we assume that there are about 100 computational operations per sample for each voice, that is 5 million operations per second per voice. An operation involves a multiplication and an addition. By a voice we mean one horn or one string of a stringed instrument. A mid-size computer of today (VAX-750) is capable of about only 250,000 arithmetic operations per second, meaning by our model that it is capable of computing only about 1/20 of a single voice. When the data-shuffling and housekeeping operations necessary to run a real instrument model are included, the factor increases another order of magnitude, so it is hopeless to compute the sounds in real time. Today's most powerful computers are capable of computing only a small number of voices.

In the past the enormous computation bandwidth of sound generation has been avoided by using musical shortcuts such as waveform table lookup and interpolation. While this approach and those built upon it can produce pleasing musical sounds, the attacks, dynamics, continuity, and other properties of real instruments simply cannot be captured. In addition, traditional methods suffer from this shortcoming, that the player of the instrument is given parameters that don't necessarily have any direct physical interpretation and are simply artifacts of the model. It would be nice, for example, to supply a musician or composer with a instrument having strings with mass, stiffness and tension that can be varied dynamically. This capability is possible if a representation of the instrument is based on its physics.

An even larger problem with the shortcut methods of the past is that they have produced models that require updates of internal parameters at a rate that is many times that which occurs in real musical instruments. The control, or update, of parameters has become an unmanageable problem.

In this thesis I present a solution to the problem of the generation of realistic musical sounds. The solution is based on using physical modeling and a VLSI implementation.

Chapter 1, *Modeling Musical Instruments*, summarizes past attempts at sound synthesis and presents an alternative approach based on physical models. Chapter 2, *Computing Sound*, presents our computational idiom, that is, a method of computing musical sounds, and describes the building blocks used to implement musical instrument models. Implementations of the models presented in Chapter 1 are described. Chapter 3, *A VLSI Architecture*, presents a computer architecture based on VLSI and our computational idiom. The details of the implementation of the architecture is presented in Chapter 4, *VLSI Implementation*, along with the introduction of new CMOS circuit techniques. In Chapter 5, *Other Architectures*, I investigate other VLSI architectures for sound synthesis.

Chapter 1

Modeling Musical Instruments

This chapter presents an approach to the synthesis of musical sounds. I begin with a brief summary of current methods of sound synthesis and of the background information for our approach. Next, I develop computational models for two representative musical instruments. The form that the models take is strongly dependent on the implementation strategy, or computational metaphor, that we have adopted.

Methods of Sound Synthesis

As background to our approach to sound synthesis I present a brief summary of popular forms of real-time generation of musical sounds with electronic equipment in use today. Other nonreal-time techniques exist and in some cases produce higher-quality results. They will not be considered in this summary because we are interested only in real-time synthesis. For a complete summary of existing sound-synthesis techniques and equipment see [Gordon 85] and [Roads 85].

Additive synthesis is a method that has been popular in experimental laboratories for many years but has only recently enjoyed commercial success because its implementation is expensive. The technique is based on the Fourier theory, which states that any signal may be analyzed and reconstructed by summing sine waves. The information that must be extracted from the original sound is the frequency, amplitude, and phase as a function of time of each sine wave. A large number of sine wave oscillators must be used to reconstruct the sound accurately, making the approach expensive. Digital technology has reduced the cost of oscillators, however. This method performs well for the steady-state and nearly periodic parts of sounds but is less successful for initial transients. Discrete Fourier analysis inherently averages the waveform in the time domain to extract exact frequency information, destroying fine-time information.

Because of the relatively high cost of additive synthesis, *subtractive synthesis* has enjoyed more success in commercial applications. The idea is to start with a wide-bandwidth signal (one containing many harmonics) and to use filters to shape

the spectrum of the sound to match that of a desired tone. The characteristics of the filter need to change in time to emulate the time-varying spectrum of the desired tone. Subtractive synthesis has worked fairly successfully in generating humanlike speech by viewing the vocal tract as a filtering of glottal pulses. However, the filters used in electronic synthesizers have been much simpler than those of the vocal tract. Also, the input waveforms used often are simple square waves, sawtooth waves, or triangular waves—signals the components of which are perfectly harmonic. The result is often a small range of effects that are characteristically *electronic*-sounding.

Sounds produced by these two techniques rarely are mistaken as originating from physical instruments. It is not clear whether, with these techniques, inventors intended to emulate pre-existing instruments, but it is certainly the case that the sounds of pre-existing instruments did provide the original inspiration. These early attempts, although failing to reproduce faithfully the sounds of pre-existing instruments, provided musicians and composers with the source of new and, in some cases, interesting sounds.

A method that has recently become popular directly attacks the problem of generating the sounds of physical instruments. Digital technology and the low cost of semiconductor memory have made it possible to store waveform information in digital form and to retrieve it for real-time playback under user control. This method, called *sampling*, has the problem that there never is and never will be enough memory to store all desired sounds. An instrument not only can play many notes but also can play each note in a variety of ways. For example, when a string is plucked hard, it is not sufficient to reproduce the sound as a loud version of a soft pluck; the perceived timbre of the tone changes with the plucking strength. A faithful reproduction of the instrument must take plucking strength and other player parameters into consideration.

To relieve the heavy memory burdens imposed by sampling, many proprietary techniques have emerged for data compression. One obvious approach is to store only the unique portions for each note—the attacks, decays, and a representative slice of the sustained portion of the note. Other techniques compress the data needed by recording only a few notes over the range of the instrument and using interpolation and extrapolation techniques to generate the other notes on demand. Similar techniques have been used to store single notes played in a variety of styles, for example, with different intensity.

FM synthesis has been revolutionary in electronic sound-synthesis. The technique, invented by Chowning [CHOWNING 73], uses frequency modulation by controlling the frequency of one sine wave with another to produce dense harmonic structure. Unlike the FM used in communications applications, where the percentage of modulation of a carrier frequency is small and consequently a small number of significant sideband frequencies are produced, in musical FM synthesis the amount of modulation is very severe, producing many sidebands. It is possible to arrange the sidebands to be harmonic or to lie in other musically interesting series, but control over each sideband individually is not possible. The growth of the sidebands or harmonics in time is controlled with an envelope on the modulating oscillator. The power of the technique is that a potentially large number of components are generated using only two oscillators. Other researchers have extended

the FM technique to modulate the frequency not of a second sine wave but of a general polynomial function [ROADS 85]. The generalized method is called *nonlinear waveshaping.* These methods, along with sampling, form the core of the techniques used in today's synthesizers.

One of the problems with FM synthesis results from the fact that nothing resembling this type of frequency modulation occurs in physical musical instruments. The technique is clever and elegant but is in some sense a trick to imitate the sound of the instrument rather than the instrument itself. Therefore, the user is not provided with a meaningful parameterization of the physical instrument. It is a complicated and sometimes impossible task to go from a physically caused phenomenon to a set of synthesis parameters. This problem is generally true of all the methods presented in the preceding discussion.

The parameterization, or control, problem is just one of a variety of problems that exist due to the fact that nowhere in the sound-synthesis model is there a model of the physical instrument. The methods described attempt to imitate the sound without imitating the instrument and in general have not been successful.

Most nonphysical sound models do not contain a representation of the state of the instrument, and as a result each note cannot depend on the history (last sequence of notes) in any way. Not only is interaction between successive notes disallowed, but also the previous one is usually terminated when a new note is begun. A dramatic example of the need to represent the state of the instrument is apparent in a repetitively struck church bell. The sound resulting from each strike is a function of the nature of the strike *and* of the state of the bell at the instant that it is struck. Some strikes sound harder and others softer, depending on the surface of the bell at the exact instant the hammer contacts the surface. In addition, because the bell has many long-lived modes, the effects of many strikes all exist simultaneously within the system. Similar effects occur in all physical instruments.

Another aspect of sound generation that has not been captured by the methods of the past are effects due to coupling between resonant members of the same instrument. Consider the piano as an example. The timbre of a struck string (or set of strings) depends not only on the state of the string and how it it is struck, but also on the state of the bridge, the soundboard, and certain other strings (the ones without a damper). All the strings couple through the bridge and soundboard, affecting the sound of a struck string and producing sympathetic vibrations. The way strings couple with one another depends on which keys are pressed (and thus which dampers are released) at the time a new key is pressed.

The human hearing system has evolved to be extremely sensitive, particularly with respect to transient behavior. The nonphysical sound-modeling techniques, except for sampling, start notes off in a very simple and pure way, contrary to the way sounds from physical instruments are started. The sounds from most musical instruments in reality begin with an almost chaotic behavior before the instrument develops coherence and produces a pure tone.

In spite of these drawbacks these techniques have enjoyed popularity for three reasons:

1. Under controlled conditions, for certain sounds, people can be fooled into hearing natural sounds.

2. The electronic instruments have developed on their own merit as new instruments with new sounds.
3. Nothing better has been available.

An obvious step is to develop a method of generating sounds by mathematically modeling the motions of the physical musical instrument. Generating sound by solving the equations of motion of an instrument captures a natural parameterization of the instrument and includes many of the musically important physical characteristics of the sound. A large literature exists concerned with mathematical modeling of musical instruments, or at least pieces of musical instruments. Most of these projects have not had the benefit of humans being able to listen to the results, and progress has been slow. Human hearing is complex and is itself poorly understood, so it is difficult to refine models without the benefit of listening tests. Weinreich is a notable exception [WEINREICH 79]. He has been primarily interested in understanding the physics of musical instruments and has used listening to synthetic sound as a way to check his theories and models. Hiller and Ruiz did work solving the wave equation for a string using finite differences on a conventional computer as a way to generate sound [HILLER 71]. Their research resulted in interesting experiments with a natural parameterization of a string, including density, elasticity, stiffness, and rigidity of string end supports.

Physical Modeling

In this section, we step back and explore the possible techniques for emulating the physical behavior of musical instruments. The conventional way of representing acoustical systems, and perhaps the most general way, is as a set of coupled partial differential equations (PDE) in time and the three spatial dimensions. If such an approach is practical, in many ways it is the ideal way to produce sound. All the problems mentioned with traditional sound-synthesis techniques are avoided naturally. In fact, few people would argue that, implementation issues aside, a PDE representation for musical instruments is the best representation; the problems arise because of the impracticality of numerical solutions to such systems. The power of the PDE technique lies in the fact that the interesting and complex behavior of most musical instruments arises from the interaction of their pieces.* The behavior of each constituent piece is close to that of an ideal theoretical case. The rich interaction among elements is also what has eluded researchers in arriving at simple closed-form solutions for such systems over a wide range of parameters.

The two popular techniques for numerically solving a system of PDEs are the methods of finite elements and finite differences. Both methods discretize the systems in the spatial and temporal dimensions. Variables are used to represent the state of each element or each discrete piece of the system. Local laws based on conservation of energy govern the interactions among the elements of the system. The system can be solved by sequentially updating the state of each piece of the

* A notable exception to this theory is the Chinese gong, the timbre of which evolves as the result of a nonlinearity in the metal due to hammering during construction.

system represented in the memory of a conventional computer, or by simultaneously updating them on a parallel computer. These methods work well for most physical systems because the behavior of the system is described very accurately by local effects. The question is: How powerful a computer do we need to model interesting instrument behavior in this fashion?

Assume that we can determine local laws governing the motion of each element based on the position and motions of its neighbors. The laws may change for various parts of the system, depending on the nature of the physical materials involved. A key questions is: How many elements do we need to represent the motion of the system accurately, and how much computation is required? For a limiting case, we will consider a grand piano, modeling it as a three-dimensional collection of elements, each representing a small piece of the system. Because we want to do a good job of modeling all the interactions within the piano, including the air in the spaces between the strings and the other pieces of the piano, we will simply slice the entire volume of space occupied by the piano into small cubes. A side effect of approximating a continuous system with discrete elements is that the resulting structure is inherently *dispersive*, even when the continuous system is not [BRILLOUIN 46]. In other words, the velocity of wave propagation is a function of the frequency. Of course, the smaller our elements and the finer the discrete approximation, the less severe the dispersive effect. A rule of thumb used in siesmology analysis that is probably sufficient here is that the smallest interesting wavelength should be represented by at least 10 elements. The smallest interesting wavelength in the system will be a high harmonic on the longest string. The longest string on a piano has a fundamental frequency of 27.5 Hz and is approximately 2 meters long, corresponding to a phase velocity of about 55 m/sec. Therefore, a frequency of 20 KHz corresponds to a wavelength of 2.75 mm. Using our factor-of-ten rule, each element needs to be 0.275 mm in each dimension. The volume of space used to represent the piano is about 2 meters long by 1 meter wide by 0.5 meters deep, or 1 cubic meter. So we need 3600 elements in each dimension for a total of about 50×10^9 or 50 billion! The time step normally chosen corresponds to a sampling rate of about 50,000 samples per second. The state of each element must be updated at every point in time according to the state of its neighbors. Therefore, the total number of updates per second is 50 billion, where each update involves several common arithmetic calculations, such as additions and multiplications. This requirement cannot be met by any computer in existence today nor probably for some time to come.

The enormous computation required for the above simple-minded approach to simulating physical behavior makes the technique impractical as a way to generate sound. A way to reduce the computational complexity of the task without giving up the physical essence of the representation is to attempt to reduce the dimensionality of the problem. Stretched strings, for example, can be approximated as one-dimensional structures, as can the air columns of woodwind and brass instruments. Accurate simulations of plates and membranes require a two-dimensional representation. Interfaces between the various elements of the system still exist, but each element takes on a lower-dimension form. Not enough work has been done in these areas, however, for us to know what is lost sonically in such approximations.

One-dimensional approximation takes one of two forms: The most obvious form is simply a one-dimensional finite-element or finite-difference approach. The finite-difference method was used by Hiller and Ruiz [HILLER 71] to simulate the motion and sound of a string. A much more efficient simulation for some computers uses the concept of wave impedance. Mediums are approximated by coupled sections of constant impedance. A forward and backward wave moves through each section unchanged but delayed in time. There is a strong correspondence between these systems and scattering theory, and the well-known ladder and lattice filter structures used in signal processing. The interface between each constant impedance section forms a scattering interface where there is a wave transmission and reflection. The form of the computation at each interface can be formulated from conservation of energy. Nondispersive media or mildly dispersive media may be implemented quite efficiently, using delay lines with computation joining them. The delay lines simulate the propagation of the wave through a section of constant impedance, and the computation joining each section computes the scattering. These systems have been explored by J. Smith in an approach to modeling he calls *waveguide digital filters* [SMITH 82]. Corrections having to do with representing a nonideal medium with an ideal one—namely, memory—sometimes can be lumped into the ends of the lines where the computations take place.

The next logical step in lowering the dimensionality of the elements is to approximate coarsely the spatial dimension by forming a *lumped* system for each element, similar to conventional linear filters or lumped electrical circuits. The instrument model that results is a system of coupled ordinary differential equations (ODE) in time. This is the approach we have taken in this work. The ODEs are approximated by finite-difference equations and are solved concurrently.

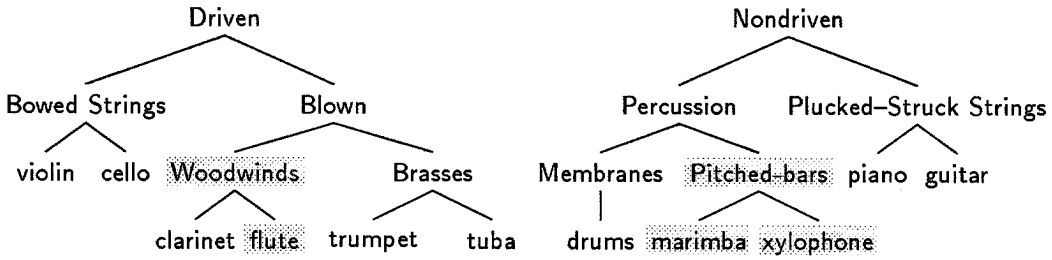

Figure 1.1 Partial taxonomy of modern orchestral instruments.

Figure 1.1 shows taxonomy of modern orchestral instruments, it is not complete and for clarity does not necessarily use the proper technical names for each category. The primary division separates *driven* instruments, those with a forcing function, from *nondriven* ones, those which are excited then allowed to sound freely. The two instrument classes differ significantly in their sound production mechanism. We have investigated one case from each of the two classes of instruments and have developed models used to compute sound. From the driven class we have chosen the flute, recorder, and organ pipe instruments, all of which share a common sound

production mechanism. From the nondriven class we have chosen the bar-percussion instruments, which with minor modifications can be extended to emulate plucked strings and struck bells and chimes.

Computational Model for Organ Pipes and Flutes

Introduction

Organ pipes, recorders, and flutes all share a common sound-production mechanism. This mechanism is fairly well understood and has been presented in the literature [FLETCHER 80], [FLETCHER 83]. As is true of our response to most other musical instruments, nonlinear effects play an important role in our perception of the generated sound. These nonlinearities make musical instruments particularly difficult systems to study analytically, for a formulation of a closed-form model of such a system requires the solution of coupled nonlinear equations. This difficulty has led to the development of time-domain models [MCINTYRE 83]. The time-domain models provide the basis for efficient numerical solutions and thus a method for simulation.

One fortunate feature of flutes and other woodwind instruments is that there are two separable systems. The nonlinear effects are essentially concentrated at the blowing mechanism and thus can be split off from the remainder of the system, leaving a linear system. This lumping of pieces of the system into distinct parts is essential to efficient simulation.

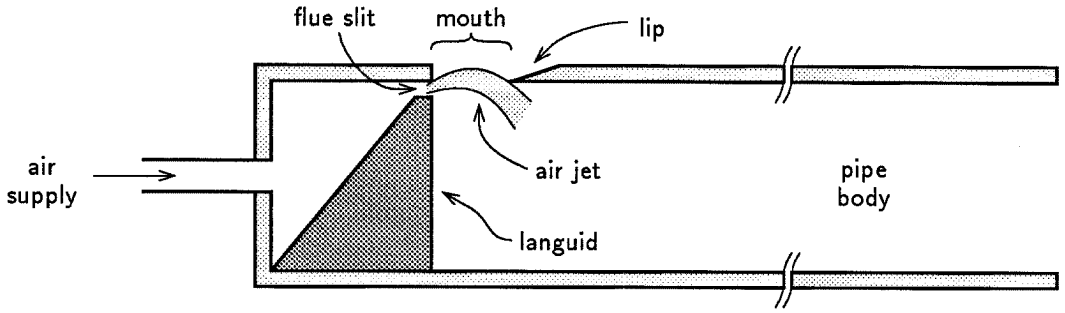

Figure 1.2 Organ pipe.

A typical organ pipe is shown in Figure 1.2. Air is forced through the *flue*, across the mouth of the pipe where it hits the *lip*. In the flute, the player's mouth and lips play the role of the flue slit. The air forced through the flue interacts with the air within the pipe body such as to generate a sustained oscillation. From basic linear-system theory, the system must contain a negative resistance (or positive feedback) if it is to sustain an oscillation. How does a negative resistance show up in a flute? This question is easiest to answer in the case of clarinets and other reed woodwind instruments, and from there it is straightforward to understand the flute.

The player of the clarinet supplies a steady source of pressure to the outside of the mouthpiece. When the player blows lightly, a small amount of air flows past the reed into the mouthpiece. As the blowing pressure is increased, the amount of air flowing into the mouthpiece increases proportionally. As the blowing pressure is increased even more, however, the reed begins to impede the flow of air and begins to close. The reed is springy, so the harder the player blows, the more the reed closes, and the smaller the amount of air that flows into the clarinet. In fact, if the player blows hard enough, the reed will close completely. If we view the reed as a resistance that converts pressure across it into air flow into the mouthpiece, then, when it is working in its springy regime, it is a *negative* resistance because, as the pressure is increased, the flow is decreased.

Now assume that the clarinet is making a sound. There is a pressure wave within the clarinet body with a wavelength proportional to the length of the body. The pressure just inside the mouthpiece on the inside of the reed is increasing and decreasing with the fundamental frequency of the tone. When the pressure in the mouthpiece is low, the pressure *across* the reed is high and the reed closes, allowing less air to flow into the mouthpiece, lowering the internal pressure even more. Conversely, when the pressure in the mouthpiece is high, the reed opens, allowing more air to flow in, increasing the pressure.* If we view the pressure wave within the body as the sum of a left-going and a right-going wave, the amount of extra pressure supplied at the mouthpiece during one cycle must be just enough to refurbish what is lost in one round trip down the pipe body and back.

In organ pipes, the essence of the oscillation mechanism is the same as it is in clarinets. The air supply is forced out the flue slit across the mouth of the pipe, and it forms a sheet of air. The sheet is a turbulent jet that reacts to the acoustical vibrations of the air within the pipe and is analogous to the reed in the clarinet. If the jet were simply allowed to move in and out with the air in the pipe, the oscillation would be canceled out. As the pressure within the pipe increased, it would force the jet out of the pipe, decreasing the pressure in the pipe. The situation is similar for a low pressure in the pipe. The air jet does not move as a flat sheet, however; the jet interacts with the acoustical vibration in the pipe, which in turn induces transverse waves on the jet traveling from the flue slit across the mouth. The distance from the flue slit to the lip on the pipe is carefully designed to correspond to one-half the wavelength of the transverse wave on the jet. The delay along the jet results in the jet's alternately blowing in and out of the mouth of the pipe one-half cycle out of phase with the acoustic displacement out of the pipe mouth due to the vibration of the air column. Alternatively, we can describe the jet as behaving as a mass termination, responding one-quarter cycle out of phase with the force acting on it (pressure) and thus one-half cycle out of phase with the displacement. This one-half–cycle delay is the source of the negative resistance.

It is interesting to look in detail at the jet flow in and out of the pipe. Even if the jet deflection at the lip is nearly sinusoidal, the jet flow saturates once it is completely

* The story is slightly complicated by the fact that the size of the opening and the amount of the wave reflected are changing as the reed opens and closes; for the purposes of this discussion, we can ignore these effects.

blowing into the pipe and similarly when it is blowing completely out of the pipe. This saturation results in an approximate hyperbolic-tangent function relating jet flow into the pipe to jet deflection. The distortion of the jet deflection by the tanh function injects odd-numbered harmonics into the pipe along with the fundamental frequency of the deflection. Of course, odd harmonics are generated for all the components of the waveform on the jet; however, it has been found by Fletcher and Douglas [FLETCHER 80] that, in practice, the fundamental is the primary component surviving the trip along the jet. The pipe body has resonant modes corresponding approximately to the harmonics of the fundamental frequency of the oscillation, and it adds gain at the frequencies of the harmonics, producing a more pleasing musical tone. Even numbered harmonics are generated by offsetting the lip of the pipe slightly from the center of the flue such that the lip cuts the jet away from its center place. This offsetting causes the waveform of the flow into the pipe to be nonsymmetric, and thus to contain even harmonics. Offsetting of the lip is used by pipe-organ builders to adjust the tonal quality of pipes, and also by flute players who direct the air jet relative to the pipe lip to achieve a desired timbre. Timbre is also adjusted by instrument builders by varying the pipe geometry. Narrow-diameter pipes have more efficient higher resonances and thus produce a brighter tone relative to wide pipes, which emphasize the lower frequencies, resulting in a duller tone. Closed-ended pipes are also sometimes used when only odd harmonics are desired.

Model Equations

Basic Model. In this section, I develop model equations for the sound-generation mechanism in the flute, suitable for simulation. From the line of modeling developed in [MCINTYRE 83] for the clarinet, I develop model equations for the organ pipe and transform them into a description suitable for direct execution within our computational metaphor.

I begin by looking at the linear part of the flute—the pipe. The signal variables of interest within the pipe are *pressure*, volume *displacement* and its time derivative *flow*, or volume velocity. Pressure and flow in pipes correspond to voltage and current in electrical transmission lines, and as electrical impedance relates voltage to current, acoustical impedance relates pressure to flow:

$$P = Z \cdot U,$$

where P is pressure in units of Pascals or $\mathrm{kg}/(\mathrm{sec}^2 \cdot \mathrm{m})$, U is flow in $\mathrm{m}^3/\mathrm{sec}$, and Z is acoustical impedance in units of $\mathrm{kg}/(\mathrm{sec} \cdot \mathrm{m}^4)$. Displacement is in units of m^3. The acoustical impedance for a uniform tube is real and positive and is simply equal to C (speed of sound) times air density divided by cross-sectional area. Assuming the pipe is a linear system, we represent the pressure wave within the pipe as the sum of an incoming wave and an outgoing wave, $P^+(x,t)$ and $P^-(x,t)$, respectively, functions of time and place along the tube. The place variable x runs from left to right in Figure 1.2. The associated flows are related to their respective pressure components by the impedance:

$$U^+(x,t) = -P^+(x,t)/Z(x,t),$$
$$U^-(x,t) = P^-(x,t)/Z(x,t).$$

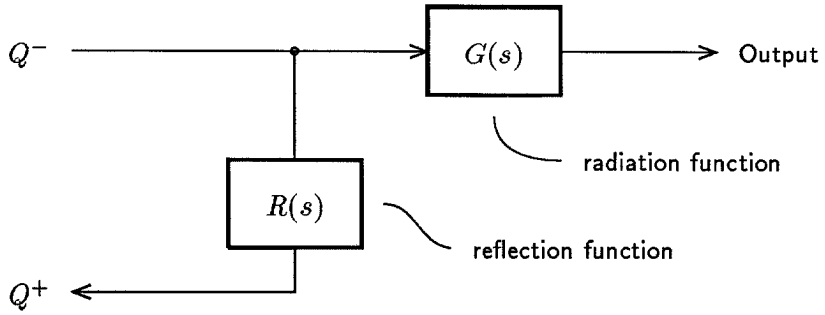

Figure 1.3 Model of flute pipe body. The radiation and reflection functions are represented by their s-transforms.

For a uniform section of pipe, Z is constant along the pipe and, barring atmospheric changes, Z is constant in time; therefore, we drop the functional form for Z.

If we fix our point of reference for P and U, say at the extreme left of the uniform pipe, then P and U are functions of time only and we replace $U^+(t)$ with U for simplicity:

$$U^+ = -P^+/Z,$$
$$U^- = P^-/Z.$$

The total pressure at a fixed reference within the pipe is

$$P = P^+ + P^-.$$

Alternatively, we can represent the displacement at a fixed point along the pipe as

$$Q = Q^+ + Q^-. \tag{1}$$

With this decomposition of the displacement wave at the left (and blown) end of the the pipe, we describe the effect of the pipe body in terms of its impulse response. An incoming wave is the convolution of the outgoing wave with the impulse response of the pipe body:

$$Q^+ = r(t) * Q^-, \tag{2}$$

where $r(t)$ is the impulse response of the flute body and is positive for an open pipe. Of course, $r(t)$ may be very complex, depending on the reflectivity properties of the flute body. The geometry of the pipe, placement and size of the tone holes, and shape of the bell (if present) all contribute to $r(t)$. In general, $r(t)$ contains some delay to account for the round-trip travel time of the pressure wave, and some high frequency attenuation. In addition to the reflection characteristics of the body as seen by the outgoing wave, the radiation properties of the body also are important when simulating the system. Tone holes complicate the issue, but for the simple case of an open-ended pipe, the transmission from the pipe end to the open air can be modeled as a linear filter, generally high pass. Figure 1.3 shows a simple but general model for the pipe body, where the transfer function of each component is labeled. In a model of the pipe body as a linear combination of normal modes, or resonances, $r(t)$ contributes by determining the relative amplitudes and damping of the modes. $r(t)$ by itself does not contain resonances, but the combination of $r(t)$ and the mouthpiece does.

Turning our attention to the air jet end of the pipe body, the equation governing the action of the air jet is continuity of flow within the mouthpiece:

$$U_J = U_P, \tag{3}$$

where U_J is the flow into the mouth of the pipe due to the jet and its interaction with the vibrations within the pipe, and U_P is the flow corresponding to the incoming and outgoing pressure waves. From the definition of flow, we obtain

$$U_P = \frac{dQ}{dt}.$$

The pipe is open at the jet and there is an end correction time, T_e, associated with the reflection of the incoming wave. Because T_e is very small compared to the period of oscillation, we will approximate dQ/dt as a difference over T_e. At the jet end of the pipe, Q^+ is reflected as Q^- with some change due to the presence of the jet; therefore,

$$U_P \approx \frac{Q^- - Q^+}{T_e},$$

and

$$\frac{Q^- - Q^+}{T_e} = U_J. \tag{4}$$

From our discussion in the introduction, we know that U_J is a function of the displacement at the end of the pipe and takes the form shown in Figure 1.4.

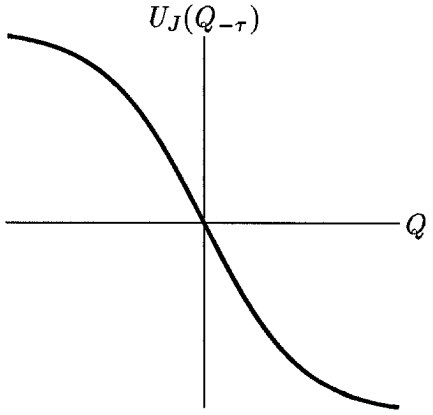

Figure 1.4 Functional form of flow into the pipe.

This curve has the form of a standard saturation curve, with the extremes representing the flow of the jet being totally in or totally out of the pipe. Implicit in U_J is a delay τ introduced by the time for the disturbance of the jet to propagate from the flue slit to the pipe lip. At resonance, τ is approximately one-half the fundamental period of oscillation. Fletcher and Douglas [FLETCHER 80] derive the functional form for the flow into the pipe for a *laminar* jet as a hyperbolic tangent. The jet of real flutes is believed to be turbulent; it is clear, however, that the actual

shape must be approximately a hyperbolic tangent. Offsetting the center of the flue slit relative to the pipe mouth is represented by offsetting the curve in Figure 1.4 in the horizontal direction.

At this point, we have everything needed to simulate the system. From Equation 4 we solve for Q^- in terms of the incoming displacement Q^+:

$$Q^- = T_e \cdot U_J(Q_{-\tau}) + Q^+. \tag{5}$$

Equations 2 and 5 are in a form suitable for computation, as shown in Figure 1.5. The radiation function of the body model is not involved in the interaction between the body and the mouthpiece and, for simplicity, is not shown.

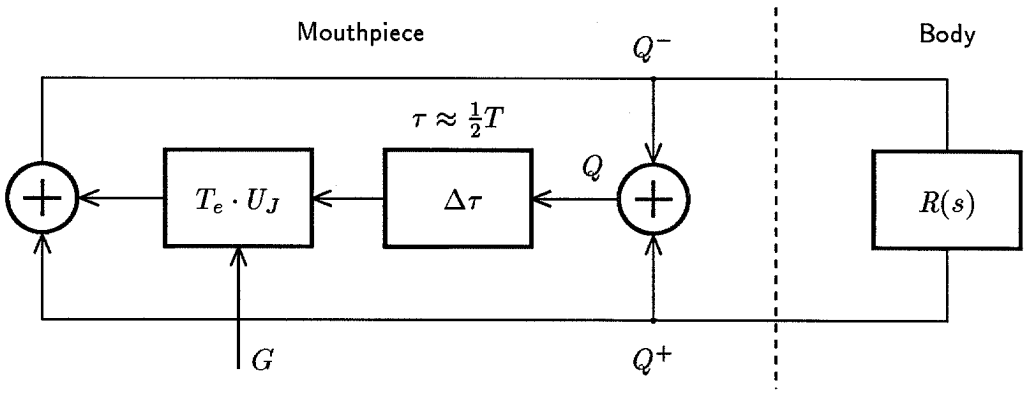

Figure 1.5 Computational model for flute.

We assume that the function U_J is scaled by a factor G such that it takes the form $G \cdot \tanh(k \cdot P)$, where G roughly corresponds to the maximum flow of the jet or the blowing strength, and k corresponds to the width of the jet. A low-flow jet saturates at a lower level and thus has a smaller G.

The following describes an interpretation of the model. When $G = 0$, the output of the functional block $T_e \cdot U_J$ is equal to zero for any input Q, and the output from the mouthpiece Q^- equals the input Q^+; in other words, the jet has no effect, and the incoming displacement wave is simply reflected off the mouth of the pipe returned. Because R is a passive function, the gain around the loop is less than unity and any disturbance simply dies out exponentially. When $G > 0$, the displacement wave emerges from the mouthpiece with amplification of frequencies near the fundamental. The saturation of the U_J curve limits the amplitude of the oscillation and also distorts the waveform, resulting in sustained oscillation and the generation of harmonics.

Noise. The model just presented captures the essence of the sound-production mechanism, including nonlinear effects and the production of harmonics, but fails to generate the noise that is present in real flute tones. It is commonly understood that the presence of noise is important to our perception of flute tones. The noise

is the consequence of turbulence in the air jet. In our abstraction of the air jet, all such turbulent behavior is lost. The exact behavior of the jet is complicated, sometimes behaving more as a laminar flow than as a turbulent flow. A detailed simulation of the air jet may be one way to achieve the desired result. However, in practice, a detailed simulation is impractical because of computational expense. A much simpler method captures much of the subjective effect of the turbulence. We observe that (1) the amount of noise injected into the pipe is proportional to the flow into the pipe, and (2) the amplitude of the signal generally grows in proportion to the amount of the flow into the pipe. Therefore, an amount of noise proportional to the amplitude of the signal is added to the signal as it leaves the mouthpiece section of the model.

Modifications. The computational model presented thus far and represented in Figure 1.5 is further simplified in this section to arrive at a system shown in Figure 1.6.

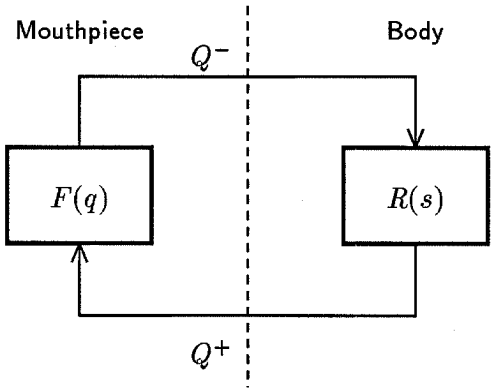

Figure 1.6 Simplified model for flute. The mouthpiece section is modeled with a single function.

The composite function of the mouthpiece, $F(q)$, is found and replaces the mouthpiece computation in Figure 1.5. Such a system is more efficient to compute in some computational metaphors, for instance, where function evaluation is done using table lookup. In addition, because the computation is in a more familiar form, it may lend itself more readily to analysis. The first simplification is to observe that the delay through the jet, τ, is approximately equal to one-half the period of the fundamental frequency, T. Thus, for a symetric U_J function, the same steady-state effect can be achieved by removing the delay and replacing $T_e U_J$ by its negative:

$$T_e U_J(Q_{-\tau}) \Rightarrow -T_e U_J(Q).$$

This replacement is inexact in that the delay along the jet actually is not constant, but is proportional to the speed of the jet, so a harder blow results in a slightly shorter delay and an increase in the frequency of oscillation. One possible solution is to add an adjustable delay in the loop.

Re-examining our Equation 5 with this modification,

$$Q^- = -T_e U_J + Q^+,$$

we find that the model equation is no longer in a form suitable for direct simulation because Q^- depends on U_J, which in turn depends on the total displacement $Q^+ + Q^-$, which includes Q^-. A similar problem arises in the modeling of reed–woodwind and bowed-string instruments. A graphical solution technique has been proposed for bowed strings in [FRIEDLANDER 53] and [KELLER 53] and applied to the clarinet in [SMITH 86]. The graphical solution effectively *presolves* the system over a range of inputs, P^+, and stores the associated outputs, P^-.

From Equations 3 and 4:

$$-T_e U_J = Q^- - Q^+.$$

Subtracting Equation 1 yields

$$\begin{aligned} -T_e U_J - (Q) &= Q^- - Q^+ - (Q^- + Q^+) \\ -T_e U_J - Q &= -2Q^+ \\ -T_e U_J &= Q - 2Q^+. \end{aligned} \tag{6}$$

Equation 6 is in a form suitable for a graphical solution technique. The procedure is illustrated in Figure 1.7.

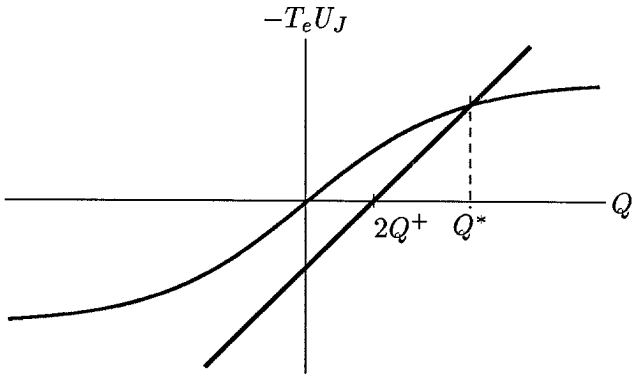

Figure 1.7 Graphical solution technique.

One curve for each half of the equation is plotted. The curve representing the left side of the equation is the scaled and negated version of the curve shown in Figure 1.4. The right side of the equation represents a line with slope of 1 and y-intercept of $2Q^+$. The value of Q at the intersection of the two curves is denoted as Q^* and represents the value of the total displacement for a particular $2Q^+$. By sweeping Q^+ and finding the intersection points, we arrive at a curve for Q^* as a function of Q^+. From the total displacement Q^*, Q^- is found from

$$Q^- = Q^* - Q^+.$$

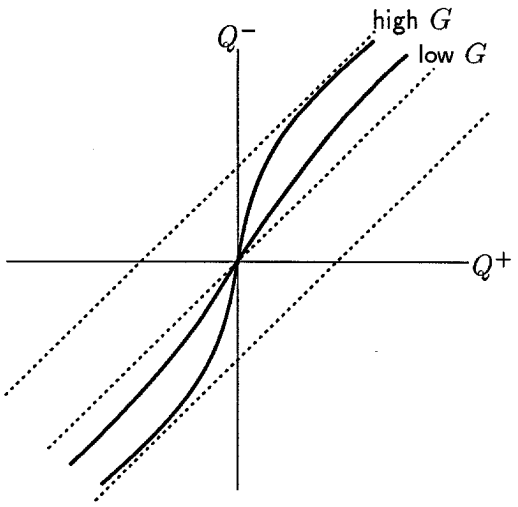

Figure 1.8 Graphical solution.

The resulting curve for Q^- versus Q^+ is shown in Figure 1.8.

The result appears much like a tanh curve rotated to 45°. The asymptote lines are at 45° and are displaced by G from a line through the origin. When the factor G goes to zero, $Q^- = Q^+$, for all Q^+.

Computational Model for Bar-Percussion Instruments

Introduction

As we saw in the previous section, blown pipes and flutes have a sound characterized by the way energy is supplied to the system. Now, we examine a class of instruments the sound of which is characterized by the way energy is dissipated. This class of instruments includes all nondriven instruments such as plucked and struck strings, drums, and bar-percussive instruments. Here I restrict the class to those instruments including a *stiff* resonant bar or plate, or those with an *elastic* member with low-amplitude oscillations. The most common instruments in this restricted class include marimbas, vibraphone, glockenspiels, bells, and chimes. Examples of a stringed instruments with low-amplitude oscillations are the Japanese koto and the Chinese guseng. Systems with elastic vibrating members such as plucked strings, where the vibration amplitude is large, include a nonlinear effect that cannot be ignored. The fundamental frequency of such systems is proportional to the *tension* on the vibrating member. The tension, however, is a function of the amplitude of the displacement of the member; as the member vibrates, it stretches, increasing its tension. The result is that the frequency is a function of the amplitude. Struck bars and tightly strung strings, on the other hand, vibrate with relatively low displacement, resulting in frequencies of oscillation independent of amplitude.

For simplicity, I will refer only to bar-percussion instruments; however, all results apply equally well to any instrument meeting the constraints outlined. All bar-percussion instruments are composed of one or more vibrating bars, a striking implement, such as a mallet, and in some cases a resonating cavity or tube to modify the sound of the bars. I will assume that, in a system of multiple bars, the bars are acoustically independent of one another. This assumption is certainly true in the case of marimbas and other such bar instruments but is not true in the case of string instruments. The strings all share a common bridge that couples energy among the strings.

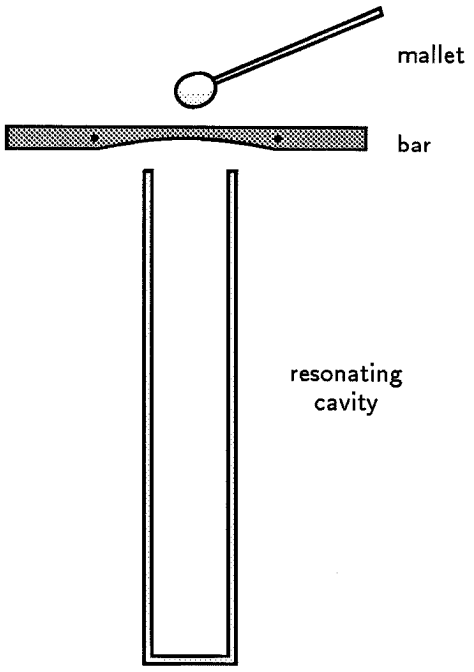

Figure 1.9 Marimba.

The timbre of the sound from a struck instrument is dependent on two sets of phenomena: (1) the physical dimensions and compositions of the constituent pieces, and (2) the manner of striking. Included in the first set is the means of support of the bar and the relative placement of the resonant cavity, if present. The manner of striking includes the force and place of the strike and can be characterized as a time-varying two-dimensional force profile on the surface of the bar. It is probably not accurate to represent the manner of striking simply as initial conditions for the bar. It is well known to students of marimba playing that the act of pulling the mallet away from the bar after a strike is important in the generation of correct timbre.

Factors influencing the sound generated from struck instruments include the following:

1. The Bar
   a. Material (e.g., wood, aluminum, steel, brass)
   b. Shape (e.g., bar, tube)
   c. Relative placement and means of support.
2. The Mallet
   a. Composition (e.g., hard rubber, felt or yarn wrapped)
   b. Size and shape
3. The Strike
   a. Position
   b. Time domain force profile.

The Bar

Ideal vibrating bars and strings have been studied for centuries. Theoretical explanations for the motion of a stiff bar and an elastic string after the initial transients have long been given in the literature [MORSE 36] and [RAYLEIGH 45]. Most authors treat bars with uniform thickness. The equation of motion of a uniform stiff bar is

$$\frac{\partial^4 y}{\partial x^4} = -\frac{\rho}{Q\kappa^2}\frac{\partial^2 y}{\partial t^2},$$

where ρ is the density, Q is Young's modulus, and κ is the radius of gyration.

The equation of motion of an ideal string is

$$\frac{\partial^2 y}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2},$$

where c is the velocity of sound on the string [MORSE 36].

The major distinction between bars and ideal strings is that the general function $f_1(x+ct)+f_2(x-ct)$ is *not* a solution for bars, and the motion cannot be represented by a sum of identical right-going and left-going waves. The velocity of propagation of a wave along a bar is a function of frequency, whereas that in an ideal string is not. This property makes bars a *dispersive* medium—a pulse sent down a bar loses its shape, because the various frequency components travel at different speeds. The consequence to instrument builders is that, even though the spatial periods of the normal modes of vibration of a bar are related closely to a series of integers, the modes do not oscillate with frequencies that lie close to a harmonic series, as they do in the case of stringed instruments.

Instrument builders have developed ways to modify the geometry of bars to force at least one normal mode to oscillate at a musical interval relative to the fundamental [MOORE 78]. Similar techniques are used by builders of violins to adjust the resonant qualities of the plates in a violin body. Builders of marimbas and other bar-percussion instruments remove material from the underside of the bar at the antinode of a normal mode, resulting in a slightly less stiff bar for that mode and a lowering in frequency. These modifications also render most of the solutions for the motions of bars of little practical significance.

Although strange geometries complicate the solution of the wave equation for bars by imposing complicated boundary conditions and violating assumptions about constants in the equation of motion, the form of the solution is not different from

that for the ideal bar. The modes are independent and have constant frequency with amplitude and time. Because of these invariants, we can model the bar as a set of independent resonances, each with a damping factor Q and center frequency θ_c. The resonances correspond to the spatial modes of the bar. Note that the resonances cannot be modeled accurately as sine waves, in spite of what the solution of the wave equation may suggest. Like those of most mechanical oscillators in nature, each resonance of the bar has a response to a forcing function, as shown in Figure 1.10.

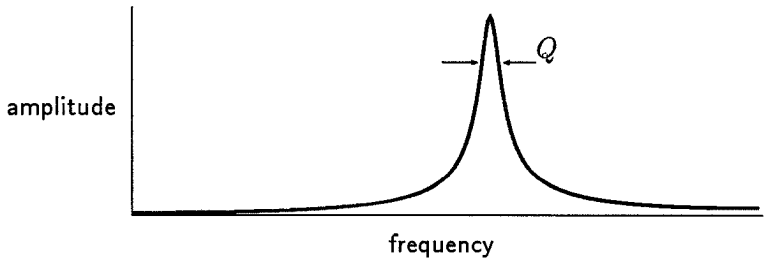

Figure 1.10 Frequency response of bar resonance.

Highly resonant modes have a narrow resonance peak. The use of simple sine oscillators loses this *band-pass* characteristic, which becomes important when considering the presence of forcing functions and the excitation of an already moving mode.

The Mallet and Strike

The bar is not modified in any way by the player of the instrument; therefore, the striking of the bar is the player's sole control over the timbre of the note. The composition and size of the mallet and force of the strike constitute the major controls used by the player. Also important to the production of a musical tone is the placement of the strike and the period of impact. How all these parameters affect the tone of the note played is fairly well understood.

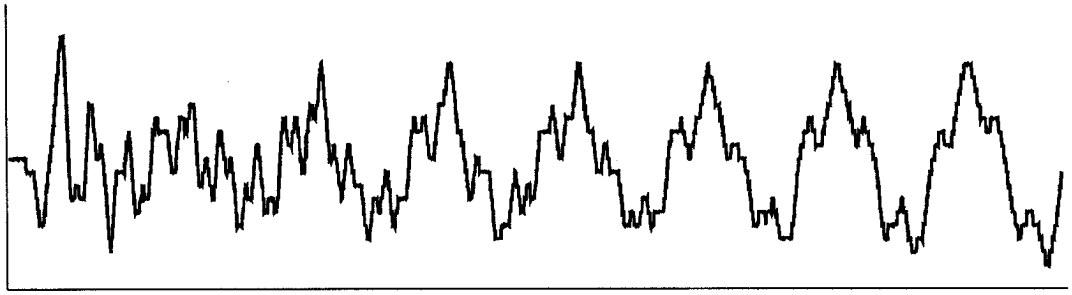

Figure 1.11 Waveform of a marimba strike. The waveform shows amplitude versus time. The bar struck was in the middle range of the instrument. It was struck with moderate force, in its center, with a hard rubber mallet.

One interesting and musically important aspect of the strike, however, is not well understood and is not treated in the literature. Even in the case of a "clean" blow to the bar, the observed waveform appears "noisy" for the first few cycles, as is evident in Figure 1.11. The noise dies out within the first few cycles. The amount of observed noise is more pronounced for forceful strikes, as well as for harder mallets. Part of the "noise" is probably due to the excitation of many low-Q, high-frequency modes. The nearly discontinuous stress profile generated on the bar propagates from the point of impact and interacts with the many degrees of freedom of the bar, generating random movements of the substance composing the bar. These random movements generate heat and are quickly damped, leaving the standing wave motion of the normal modes. Although the details of the noise-generation mechanism is not modeled by the equation of motion for the bar, the noise is critically important to our perception of the struck sound and is included in our model. The noisy beginning of the bar's motion may be viewed as an excitation function (forcing function) for the resonances of the bar. To model the qualitative effect of the noise, we need to generate an excitation function containing noise that has an amplitude proportional to the force of the strike and to the hardness of the mallet. It also is important that the noise die out after several cycles.

The key to the qualitative affects of mallet size and hardness lies in the stress generated on the bar for each case. Our hypothesis about the important affects of mallet size and hardness can be viewed as a *footprint.* As we move from a smaller to a larger mallet, the stress footprint on the bar is enlarged, providing relatively less energy to higher spatial modes. The larger the mallet, the more total energy is transferred to the bar, because a larger mallet has more mass and thus generates more force. The hardness of the mallet also effects the size of the stress footprint on the bar, because of the compressibility of the mallet. Perhaps more important, the hardness of the mallet effects the *steepness* of the stress profile into the bar. Harder mallets generate a steep stress profile and provide more energy to higher modes—and generate more noise.

Translation from stress in the spatial domain to the temporal-frequency domain is not straightforward. I use a model that captures the essential qualitative behavior with sufficient parameters to generate musically important control.

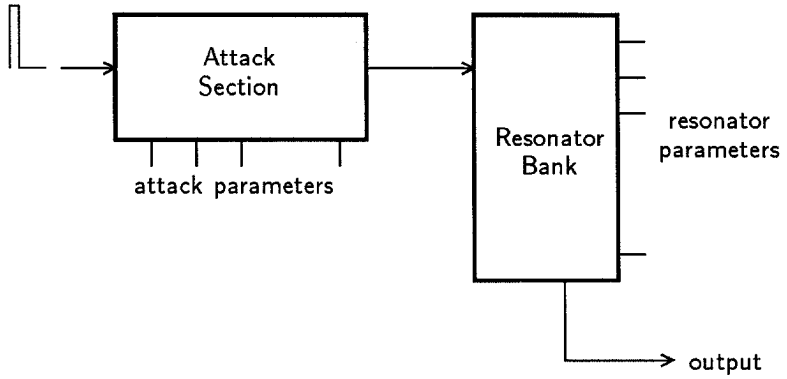

Figure 1.12 Struck instrument model.

The Model

The struck-instrument model is composed of two sections, as shown in Figure 1.12. The right circuit models the normal modes of the bar with discrete resonances. The circuit used for each mode is called a *resonator*. A weighted sum of resonator outputs is formed by summing a weighted value from the output of each resonator circuit. All the resonators are excited by a common signal from the attack section. The attack section generates a noisy signal to excite the resonators. Parameters of the attack section are changed to model different types of mallets and different force strengths. Details of the struck-instrument model with several parameters useful for specific musical instruments are presented in the next chapter.

Chapter 2

Computing Sound

This chapter presents the implementation of musical-sound synthesis. I present the implementation strategy, or computational metaphor, that we have adopted, and demonstrate it with several small examples that become the basis for our instrument models. A detailed section is presented on one particularly important computational form, the digital resonator. Finally, I present in full detail the implementation of the two musical instruments developed in Chapter 1.

Implementation Strategy

Our approach to generating musical sounds involves solving difference equations in real time. Musical instruments are modeled as systems of coupled difference equations. A natural architecture for solving systems of finite-difference equations is one with an interconnection matrix between processors that can be reconfigured (or programmed), as illustrated in Figure 2.1. A realization of a new instrument involves reconfiguring of the connection matrix between the processing elements, as well as configuring connections to the outside world both for control and for updates of parameters.

Processing elements are placed together to form an array and then are joined by a reconfigurable interconnection matrix. A general-purpose computer supplies updates of parameters to the processing elements and provides an interface to the player of the instrument. The external computer also supplies the bit patterns for the interconnection matrix. Synthesized signal outputs go to a digital-to-analog converter.

To implement a reconfigurable connection matrix, a bit-serial representation of samples facilitates the use of single-wire connections between computational units, drastically reducing the complexity of implementation. In fact, a bit-serial implementation makes the entire approach possible.

Bit-serial implementations also have the advantage in that computational elements are small and inexpensive. One potential drawback with bit-serial systems is

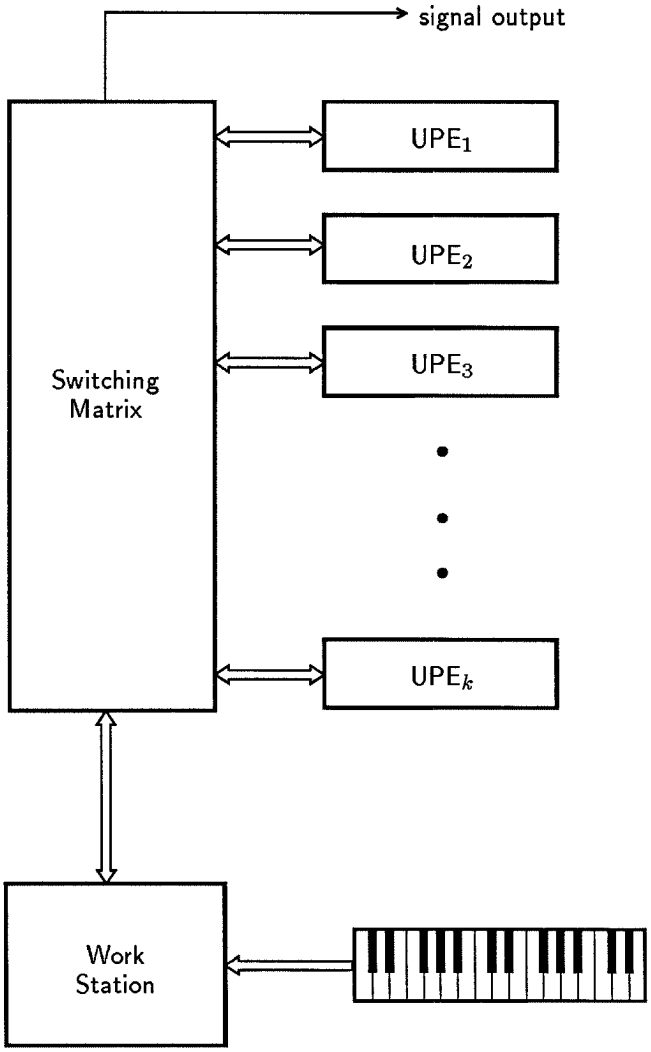

Figure 2.1 Sound-synthesis architecture. Processing elements are connected to each other and to the outside world through a reconfigurable interconnection matrix.

that they must run at a clock rate that is higher than that of their parallel counterparts. In our implementations, even with 64-bit samples, the bit clock rate is only 3 MHz, which is far below the limits of current integrated circuit technology.

For our computation we have chosen a basic unit we call a Universal Processing Element (UPE) [Wawrzynek 85a] that computes the function

$$A + (B \times M) + D \times (1 - M). \tag{1}$$

It is similar to the bit-serial multipliers proposed by Lyon [Lyon 81]. In its simplest mode of computation, where D=0, the function of a UPE is a multiplication and an addition. This simple element forms a digital integrator that is the basic building

block for solving linear difference equations. If D is not set to zero, the output of the UPE is A plus the linear interpolation between B and D, where M is the constant of interpolation.

All the inputs and outputs to the UPE are bit serial. UPEs can be connected to each other with a single wire.

Basic Elements

General Linear Filter. An M^{th} order linear difference equation [OPPENHEIM 75] can be written as

$$y_n = \sum_{i=0}^{N} a_i x_{n-i} + \sum_{i=1}^{M} b_i y_{n-i}, \tag{2}$$

where x_n is the input at time sample n; y_n is the output at time sample n; and the coefficients $a_0 \cdots a_N, b_1 \cdots b_M$ are chosen to fulfill a given filtering requirement. The function is evaluated by performing the iteration of Equation 2 for each sample time. This is the general form of a linear filter; any linear filter can be described as a special case of Equation 2.

Figure 2.2 illustrates a UPE network that directly implements the general linear-filter equation.

Each UPE (with $D = 0$) performs the function $(A + M \times B)z^{-1}$; i.e., multiplication, addition, and one unit of delay. In Figure 2.2, the input values are processed by distributing the input signal x to each of $N + 1$ UPEs, each one of which multiplies the input by a filter coefficient a_i, sums the result of the last UPE, and passes on the total for further processing. Each UPE provides one sample of delay, so the signal at the output of the input processing section is

$$X = a_0 x_{n-1} + a_1 x_{n-2} + a_2 x_{n-2} + \cdots + a_M x_{i-M+1}. \tag{3}$$

This result is summed with the result of the output processing section.

The output y_n is distributed back to each of M UPEs. Each UPE multiplies the output by a filter coefficient b_i, provides one unit of delay, sums its result with that of the last UPE, and passes on the total. The result at the end of the output processing section is

$$y_n = b_1 y_{n-1} + b_2 y_{n-2} + \cdots + b_N y_{n-N} + X. \tag{4}$$

We add the result of the input processing section to the result of the output processing section by feeding it into the UPE holding the b_n coefficient; its A (addend) input is not used. Adding the result from the input processing section to the UPE holding the b_n coefficient has the effect of adding a net delay through the system equal to the number of UPEs in the output processing section.

From Figure 2.2, it is clear that the number of UPEs needed to implement Equation 2 is equal to the number of coefficients in the input (nonrecursive) processing section plus the number of coefficients in the output (recursive) processing section.

Second-order Section. As an example of a linear filter, consider the second-order linear difference equation:

$$y_n = \alpha y_{n-1} + \beta y_{n-2} + x_n. \tag{5}$$

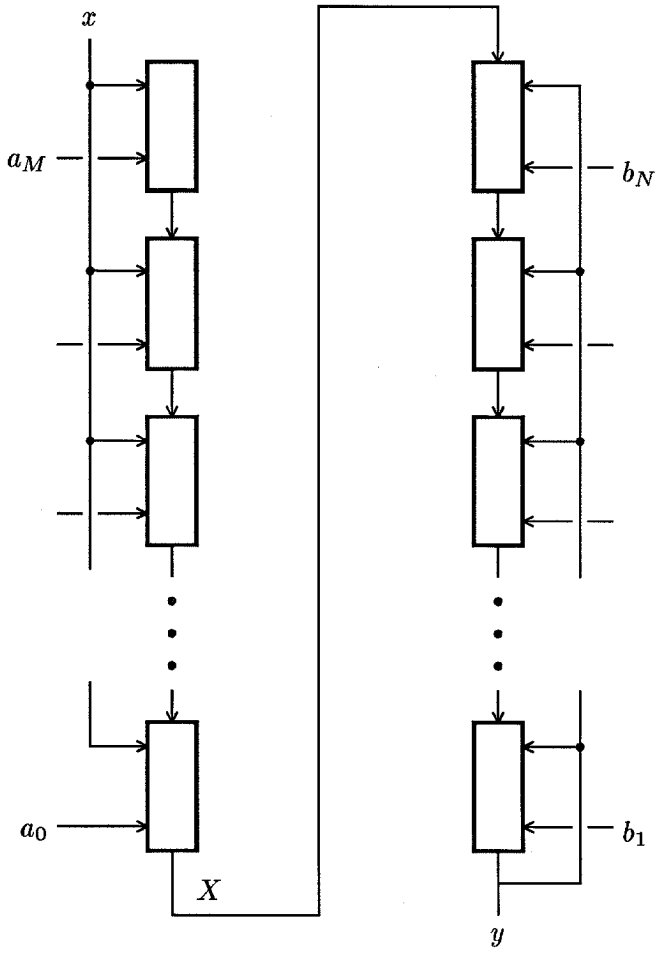

Figure 2.2 UPE implementation of general filter.

Applying the z-transform, we form the system function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{1 - \alpha z^{-1} - \beta z^{-2}}. \tag{6}$$

We have used this system widely to model the resonances of musical instruments. The theory of the digital resonator is presented in a later section. Here we will simply note that, for all commonly used values of α and β, Equation 5 can be rewritten as

$$y_n = 2R\cos\theta_c y_{n-1} - R^2 y_{n-2} + x_n. \tag{7}$$

This equation models a resonant system with damping controlled by R and frequency controlled by θ_c.

The digital resonator is implemented directly, using two UPEs. As shown in Figure 2.3. The left UPE computes:

$$(-R^2 Y + X)Z^{-1}.$$

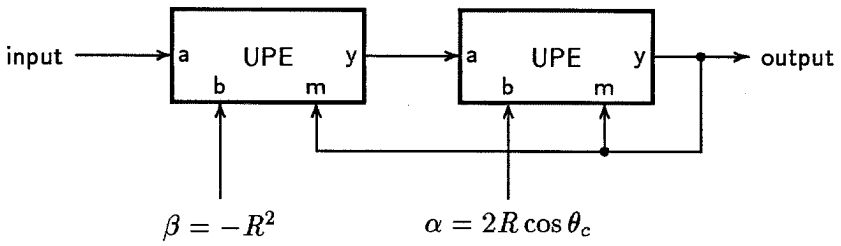

Figure 2.3 UPE implementation of second-order section.

The right UPE computes

$$[2R\cos\theta_c Y + (-R^2Y + X)z^{-1}]z^{-1} = 2R\cos\theta_c Y z^{-1} - R^2 Y z^{-2} + X z^{-2}.$$

Hence,

$$y_n = 2R\cos\theta_c y_{n-1} - R^2 y_{n-2} + x_{n-2}.$$

In the final form of the difference equation, the input variable x appears as x_{n-2} rather than x_n, as in Equation 5. This results in a frequency-independent, or *pure*, delay through the system of 2 word times. This pure delay is equal to delaying x by 2 word times before feeding it to a system as in Equation 5.

Nonlinear Element. The functions computable by UPEs are not restricted to linear ones. Certain phenomena in nature are best modeled as nonlinear functions. For example, consider the class of functions that relate flow to pressure at the mouthpiece of a blown musical instrument. A function that is characteristic of flutelike instruments is shown in Figure 2.4(c). This function and its variations, shown in Figure 2.4(a) through 2.4(d), are computed using three UPEs, as is shown in Figure 2.5. The input signal x is sent to u_1, which multiplies x by itself, creating a squared term. This same technique is used again to arrive at the function

$$y = k_0 + k_2k_3 + k_3Gx + k_2x^2 + Gx^3,$$

which is a third-order polynomial. For $k_0 = 0$ and $k_3 = -1$, the coefficient G controls the nonlinear gain, as illustrated in Figure 2.4(c) and 2.4(d). The coefficient k_2 controls the symmetry about the vertical axis, as shown in Figure 2.4(a) through (c).

This technique of generating polynomials can be extended to produce polynomials of arbitrarily high degree.

Integrator. A simple configuration using one UPE forms a digital integrator. The output is fed back to the A input, and the B and M inputs are controlled externally, as shown in Figure 2.6(a). The computation performed is

$$y_n = B \times M + y_{n-1}.$$

At each step in the computation, the product $B \times M$ is summed with the result of the previous step. This arrangement produces a ramp function, the slope of

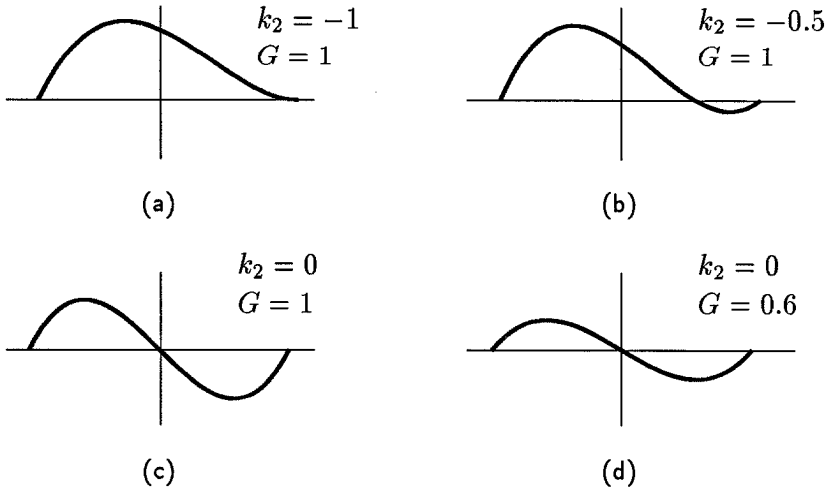

Figure 2.4 Variations of the nonlinear function.

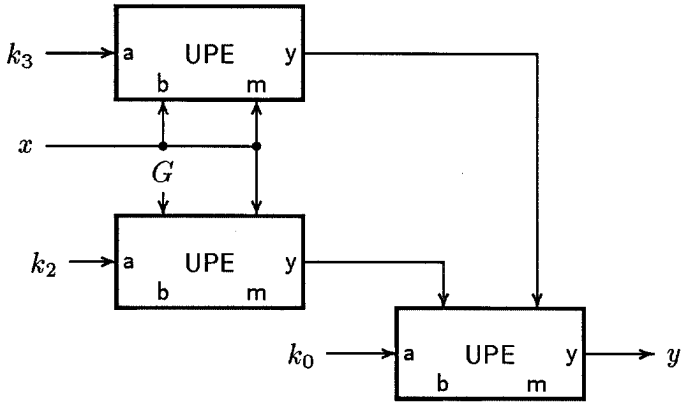

Figure 2.5 Nonlinear element implementation.

which is the product $B \times M$. As the computation proceeds, the output y_n eventually overflows the number representation and wraps around to a negative number, and the computation continues. The waveform for constant B and M is drawn in Figure 2.6(b).

Frequency Modulation. Because of the discontinuity, the ramp signal is not bandlimited, and therefore cannot be used directly for sound synthesis without aliasing components. However, in a scheme suggested by Lyon, we can remap the signal by passing it through a function, such as the one nonlinear function in Figure 2.4. When the remapping function is equal at the extremes of the number representation, as in the third-order polynomial presented previously, the resulting waveform is continuous. The resulting signal does not have the aliasing problems of the ramp function and can be used directly for musical sound application.

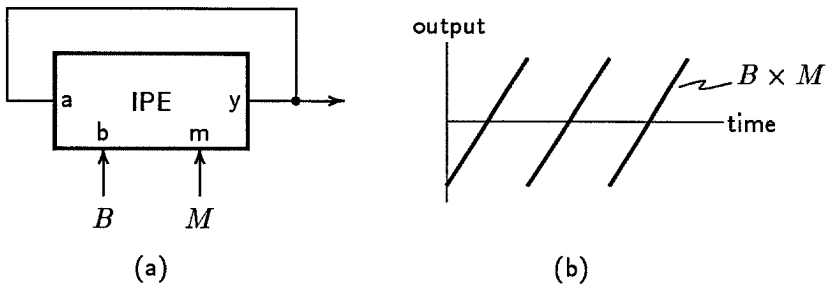

Figure 2.6 Integrator.

In this composite system, where the ramp output feeds the nonlinear section, as shown in Figure 2.7, the $B \times M$ input to the ramp can be thought of as controlling the phase of some periodic function y, and is either positive or negative. Because the $B \times M$ input can be a signal generated by another arrangement of UPEs, frequency modulation (FM) may be attained. The function generated by the nonlinear element is the carrier signal and the signal fed to the $B \times M$ input is the FM signal. This scheme is therefore equivalent to the waveform-table lookup techniques commonly used in conventional computer-music programs.

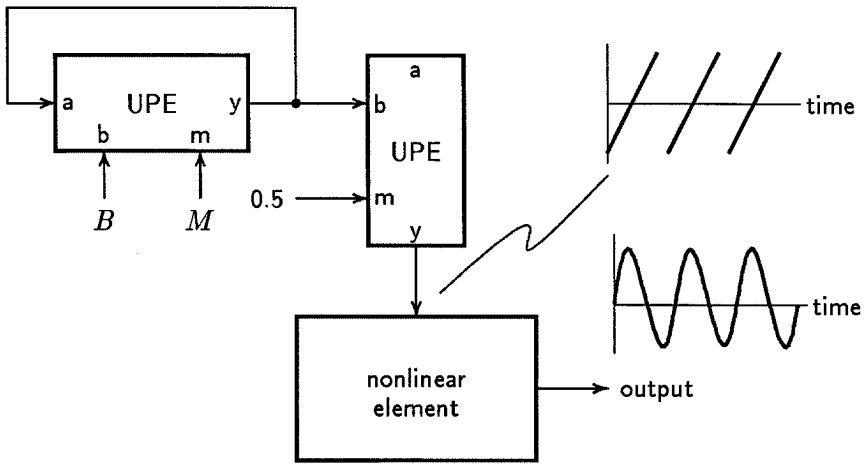

Figure 2.7 Frequency modulation.

Noise. Random signals find frequent application in sound synthesis. A pseudo–random-number generator can be constructed with one UPE, as shown in Figure 2.8. This approach uses a linear-congruence method [KNUTH 68], implementing

$$x_n = p \cdot x_{n-1} \bmod r + q,$$

where

$$r = 2^{32}.$$

The mod r operation is achieved by feeding the 64-bit output, Y, into the 32-bit input, B. Only the low 32 bits of Y get loaded, which effectively generates mod 2^{32}.

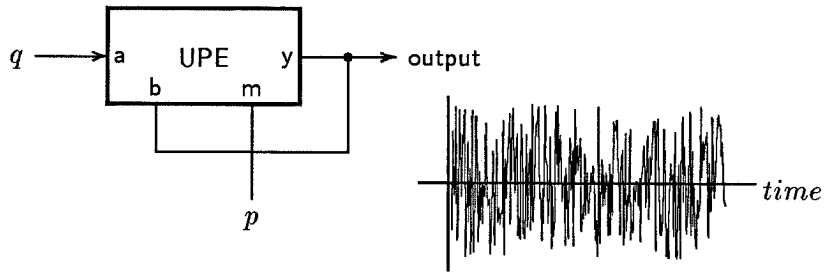

Figure 2.8 Random-number generator.

Mixer. The linear-interpolation feature of the UPEs can be used for mixing signals. Referring to Figure 2.9, one signal is fed into the B input and another into the D input. The M input controls the relative balance of the two signals in the output signal. This approach has the advantage over other schemes in that the output level is held constant as the relative mix of the two input signals is changed.

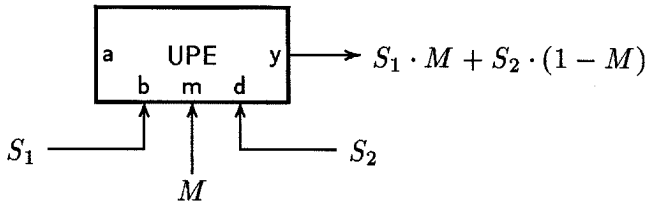

Figure 2.9 Mixing signals.

Scattering Interface. We know that the computation performed by one scattering interface is:

$$P_1^- = kP_1^+ + (1-k)P_2^+,$$
$$P_2^- = -kP_2^+ + (1+k)P_1^+,$$

where P^+ and P^- represent incoming and outgoing pressure (or force) waves, respectively, and k is a variable relating the characteristic impedances on either side of the interface [SMITH 82]. The sum $P_1 + P_2$ represent the total pressure on either side of the interface.

Each equation involves the linear interpolation of P_1^+ and P_2^+, and is computed with one UPE, as illustrated in Figure 2.10.

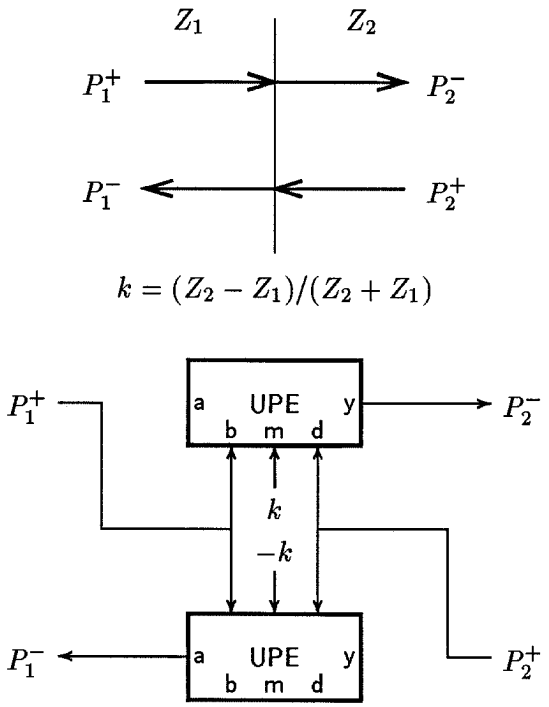

Figure 2.10 UPE implementation of scattering interface. Each UPE computes one of the scattering equations.

The function computed by the UPE network in Figure 2.10 includes one word of delay in addition to the preceding equations. In a scattering model of wave-guide filter applications, the extra delay is absorbed by shortening the delay lines between the interfaces by one word. In ladder filters, the word of delay is part of the function of one filter section.

The Digital Resonator

This section presents the basic theory of second-order digital resonators. Special emphasis is placed on those issues involving digital resonators used for musical sound synthesis. First, I present the basic equations governing the behavior of two-pole sections, including their uses in musical sound synthesis.

In the appendix, I discuss Q calculation, as a user parameter, the problems with resonator gain and a proposed solution, and the special case of critically damped sections.

Basic Equations

A digital resonator can be formulated with the finite-difference equation:

$$y_n = \alpha y_{n-1} + \beta y_{n-2} + x_n, \tag{1}$$

where x_n is the input at time sample n; y_n is the output at time sample n; and α and β are parameters chosen to produce the desired behavior. The UPE implementation of a digital resonator shown in Figure 2.3 contains a pure delay of two word times, that can be represented in Equation 1 as x_{n-2}. This delay has no effect on the amplitude of the frequency response of the system and simply delays the time domain response by two word times. Applying the z-transform, we form the system function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1}{1 - \alpha z^{-1} - \beta z^{-2}}. \tag{2}$$

Solving for the roots of the demoninator leads to two cases.

Case 1 ($\alpha^2 + 4\beta \leq 0$). In this case, the poles of $H(z)$ are complex conjugates. They appear in the z-plane at $z = Re^{j\theta_c}$ and $z = Re^{-j\theta_c}$ as shown in Figure 2.11. Here $\theta = 2\pi \times freq/f_s = \omega T$, where $f_s = 1/T$ is the sampling frequency. R is the radial distance of the poles from the origin in the z-plane and θ_c is the angle off the real axis.

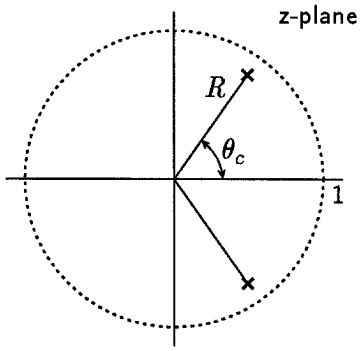

Figure 2.11 Second-order resonator poles. The R value controls the damping and θ_c controls the resonant frequency.

Now we can rewrite Equation 2 as:

$$H(z) = \frac{1}{(1 - Re^{j\theta_c} z^{-1})(1 - Re^{-j\theta_c} z^{-1})}. \tag{3}$$

Multiplying out the denominator, we get:

$$H(z) = \frac{1}{1 - 2R\cos\theta_c z^{-1} + R^2 z^{-2}}. \tag{4}$$

Rewriting Equation 1 yields:

$$y_n = 2R\cos\theta_c y_{n-1} - R^2 y_{n-2} + x_n. \tag{5}$$

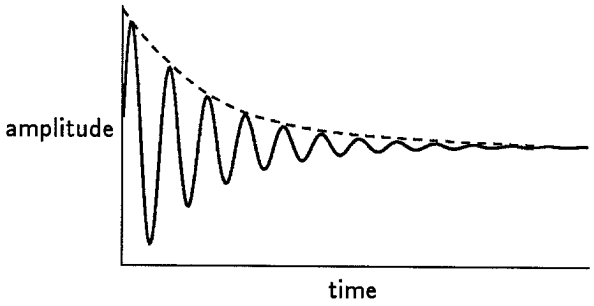

Figure 2.12 Time-domain impulse response of Case 1.

Equation (4) leads to a sinusoidal time domain impulse response of the form:

$$\gamma R^n \sin[n\theta_c + \phi], \tag{6}$$

where $\gamma = 1/\sin\theta_c$ and $\phi = \theta_c$ from the partial fraction expansion of Equation 4. For values of $R < 1$, the response is a damped sine wave with R controlling the rate of damping and θ_c controlling the frequency of oscillation.

It is interesting to note that, with $R = 1$, the impulse response is a sine wave of constant amplitude.

The system frequency response can be found by substituting $e^{j\theta}$ for z in $H(z)$. At $z = e^{j\theta}$, $H(z)$ is identical to the discrete Fourier transform. The digital resonator acts as a band-pass filter in this case, with center frequency of θ_c and a bandwidth proportional to R.

Figure 2.13 shows the magnitude and phase of the frequency response of Equation 4 for a typical set of coefficients. Note the high slope of the phase response around resonance.

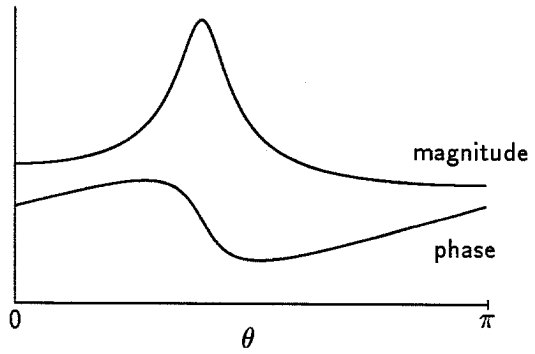

Figure 2.13 Magnitude and phase of frequency response of Case 1.

Case 2 $(\alpha^2 + 4\beta > 0)$. In this case the poles are both real, they appear on the real axis in the z-plane. The system is called *overdamped* and does not oscillate.

Application to Sound Synthesis

In our musical applications, digital resonators have found uses as a mechanism for modeling the modes of vibration (or resonances) of musical instruments. Resonators are commonly connected in the parallel arrangement shown in Figure 2.14, called a *resonator bank*. Each resonator shares a common input source, with the outputs being summed.

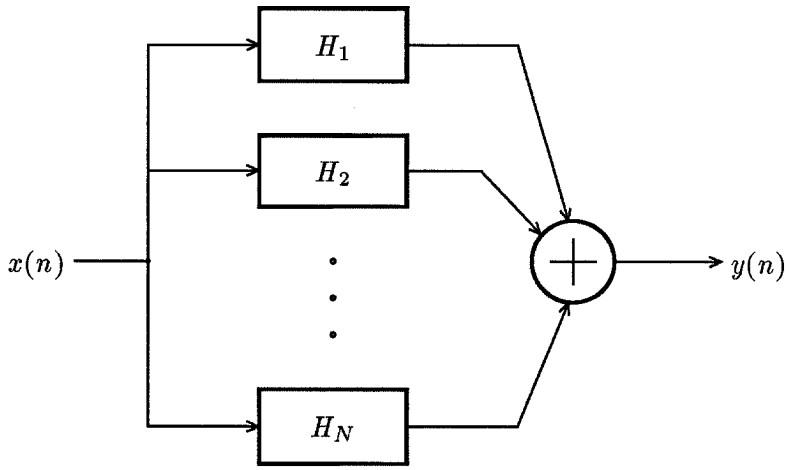

Figure 2.14 Resonator bank. Parallel connection of resonators.

The coefficients of each resonator are chosen to model some mode of the musical instrument.

In this arrangement, the system function is

$$H(z) = \sum_{i=1}^{N} H_i(z)$$
$$= \frac{1}{(1 - Re^{j\theta_1}z^{-1})(1 - Re^{j\theta_1}z^{-1})} + \frac{1}{(1 - Re^{j\theta_1}z^{-2})(1 - Re^{j\theta_2}z^{-1})} + \cdots$$

Here the poles of the system are the sum of the poles of all the individual sections. However, the numerator is no longer equal to one, and $2N - 2$ zeros have been introduced into the system. For sufficiently large values of R (low damping), the zeros have little effect on the system response. In this case the parallel connection can be viewed as a set of independent resonators.

Another common use of digital resonators in a musical context is to excite a single resonator with an impulse to exploit it as a sine-wave generator with variable damping and frequency. The sine wave is fed to another part of the musical-instrument model for further processing.

Musical-Instrument Models

This section describes two simple musical instrument models based on UPEs. Both models have been implemented and we have used them to generate musical sounds. Although these models have been used to produce extremely high-quality timbres of certain instruments, they are certainly not capable of covering the entire range of timbres of the instruments. The development of a new timbre can be thought of as building an instrument, learning to play it, and then practicing a particular performance on it. This activity requires a great deal of careful study and may involve extensions or modifications to the model.

Struck Instrument

Struck or plucked instruments are those that are played by displacing the resonant element of the instrument from its resting state and then allowing it to oscillate freely. Tone quality in such instruments is a function of how the system is excited, and of how it dissipates energy. Examples of plucked and struck instruments include plucked and struck strings, struck bells, and marimbas.

Figure 2.15 illustrates a struck-instrument model implemented with UPEs. The model can be decomposed into two pieces: the *attack section* and the *resonator bank*. The attack section models the impact of the striking or plucking device on the actual instrument. An impulse is fed to a second-order section that is tuned with a Q value close to critical damping. A detailed version of the attack section is shown in Figure 2.16. In this figure, the output of the *attack resonator* is fed to the input of the *noise-modulation section*. The noise-modulation section generates the function

$$y = NM \cdot x \cdot RNG + SG \cdot x,$$

where RNG is the output of a random-number generator. This computation adds to the signal input x an amount of noise proportional to the level of x. The balance of signal to noise is controlled by the ratio $SG : NM$, and the overall gain is controlled by $SG + NM$.

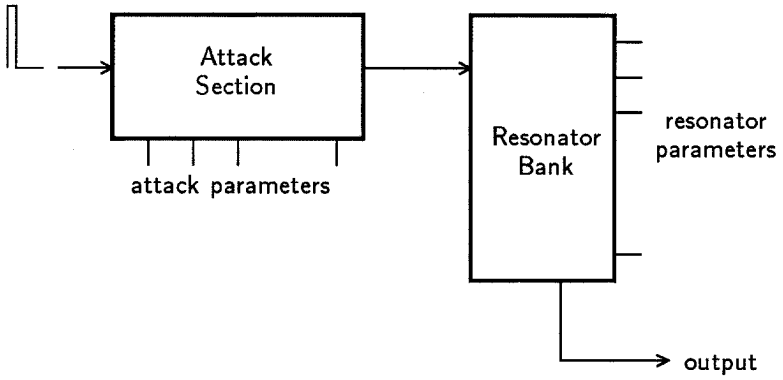

Figure 2.15 Struck instrument implemented with UPEs.

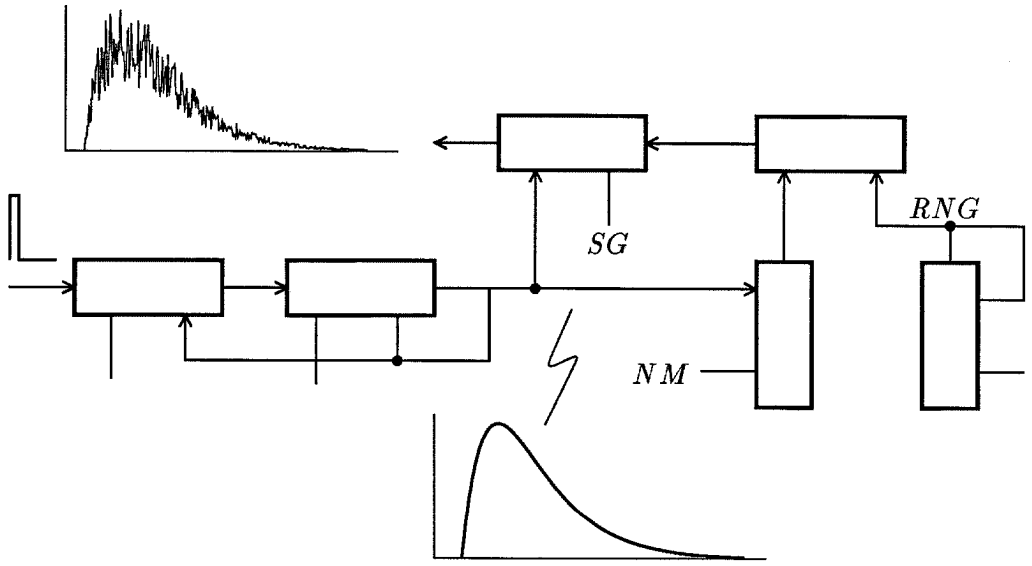

Figure 2.16 Attack section.

The output of the noise-modulation section is used to drive a parallel connection of second-order sections used as resonators. The resonators are tuned to the major resonances of the instrument being modeled. The parameters of the attack section—attack resonator frequency and Q value, signal-to-noise ratio, and attack level—are all adjusted to produce a variety of musical timbres.

Second-order sections are combined to form a resonator bank, as shown in Figure 2.17. All resonators are implemented as described in Chapter 2, *Computing Sound*, the section *Basic Elements*. The output of each resonator is connected to a single UPE that scales the output of the resonator and adds the signal to the signal from the other resonators.

The gain at resonance of a two-pole second-order section varies drastically over the frequency range. This variation causes scaling problems when fixed-point arithmetic is used. Either the input to or the output from each resonator must be adjusted to compensate for the implicit gain of the resonator.

Several techniques exist for normalizing resonator gain. One proposed by Smith and Angell uses the addition of two zeros to the second-order system function [SMITH 82]. By placing a zero at $\pm\sqrt{R}$, we can eliminate the dependence on θ in the system function. Resonator-gain normalization could pose a particularly severe problem in the case of resonator banks, as shown in Figure 2.17. Scaling the input to each resonator increases the number of UPEs by a factor of one-third and increases the control bandwidth by the same amount. Alternatively, the input to the entire system can be scaled down, to avoid overflow in the section with the most gain, and then the output scaled up to the appropriate level. This approach is a problem in systems that use fixed-point arithmetic because the amount of gain available at each multiplication is limited, and hence many multiplier stages at the output must be used.

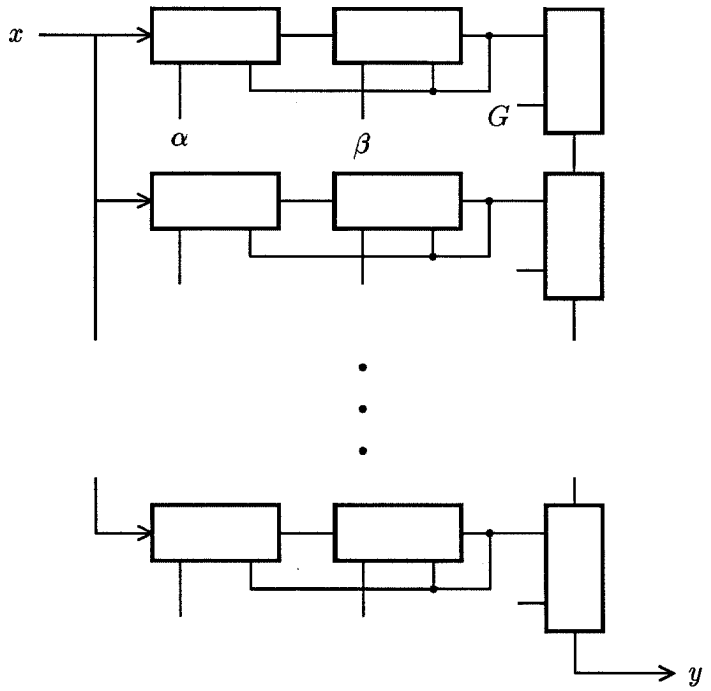

Figure 2.17 Resonator bank implementation.

In many sound-generation applications, the R values of each stage in the resonator bank are close to one another. Therefore, it is possible to synthesize two zeros using an average value for R and then distributing the result to each resonator.

In a typical application, a pianolike keyboard is used to control the instrument. The pressing of a key triggers the following actions: (1) the key position determines the coefficients loaded into the resonator bank; (2) the key velocity controls the level of the coefficient NM in the attack section (higher key velocities correspond to more noise being introduced into the the system and hence a higher attack level), and (3) the key press generates an impulse that is sent to the attack resonator.

Table 2.1 shows the parameters developed to synthesize the sound of a struck aluminum bar suspended on two loops of string at a distance of one-quarter of its length in from each end. The bar is similar to the ones used in vibraphones, without the arch cut in its underside. The bar was struck with moderate force, in its center, with a hard rubber mallet. The gain of the attack resonator is the coefficient SG. EQ_R and EQ_G refer to the two coefficients of the resonator normalization circuit. The frequency, Q, and gain of each resonator were found empirically, using spectrum analysis of the physical bar. The bar has length of 211 mm, width of 37.5 mm, and thickness of 9.5 mm. It was found to have many normal modes. Only the 10 most prominent modes were included in the simulation. Under normal listening conditions, the synthesized sound was indistinguishable from that of the physical bar.

Table 2.1 Aluminum bar synthesis parameters.

Resonator	Frequency	Q	Gain
1	1077	2000	1.0
2	2160	500	0.7
3	2940	500	0.7
4	3220	500	0.6
5	3520	500	0.4
6	3940	2000	0.4
7	5400	500	0.3
8	5680	2000	1.0
9	6900	2000	1.0
10	7840	500	1.0
attack	2000	0.5	0.004

$EQ_R = 0.0$
$EQ_G = 1.0$
noise gain = 0.0004
impulse value = 1.0

Table 2.2 shows parameters developed by Dyer for a struck marimba. The effect of the resonating cavity under the bar is incorporated into the parameters for the resonators that model the normal modes. As with those for the aluminum bar, the parameters for the resonators were found empirically, using spectrum analysis of recorded marimba sounds. Again, the strikes were of moderate force, in the center of the bar, with a hard rubber mallet. Although the parameters are shown for one particular bar (middle C), with one particular mallet type, and one particular strike, we have generalized them to simulate all the bars of the marimba, as well as other mallets, and other strike forces. We performed generalization by devising functions that scale the parameters according to user input (for example, key position and velocity) and additional input (such as mallet hardness). The scaling covers the full range of a normal marimba and also allows for experimentation with fanciful marimbas, for example, those that extend well beyond the the normal range of a physical marimba, and those with mallets that change size automatically to match better the size of the bars.

Figure 2.18 shows the first few cycles of the waveform generated by the parameters in Table 2.2. It compares favorably with the waveform of a recorded marimba strike (Figure 1.11), and with non-expert listeners in an informal listening environment it sounds virtually indistinguishable from a recorded marimba strike.

Solving the wave equation for an ideal string clamped at its ends yields normal modes, the frequencies of which are integer multiples of the fundamental. The parameters for the resonators for a plucked-string sound in Table 2.3 are based on this idea. The parameters for the attack section were found by trial and error. The resulting sound is that of a plucked, tightly strung string.

Table 2.2 Marimba synthesis parameters.

Resonator	Frequency	Q	Gain
1	261.63	240	1.0
2	1020.36	200	1.0
3	26163.0	150	1.0
attack	261.63	0.5	0.05

$EQ_R = -1.0$
$EQ_G = 1.0$
noise gain $= 0.025$
impulse value $= 1.0$

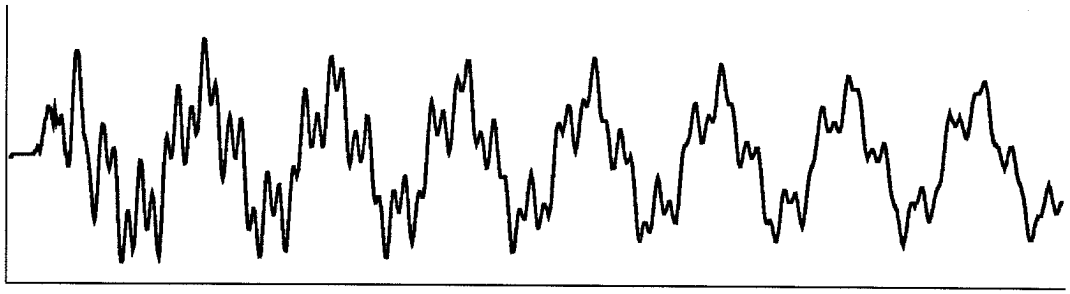

Figure 2.18 Synthesized marimba strike. The waveform shows amplitude versus time. The synthesis parameters are from Table 2.2.

Table 2.3 Plucked string synthesis parameters.

Resonator	Frequency	Q	Gain
1	440	300	0.70
2	880	300	0.80
3	1320	300	0.60
4	1760	300	0.70
5	2200	300	0.70
6	2640	300	0.80
7	3080	320	0.95
8	3520	300	0.76
9	3960	190	0.87
10	4400	300	0.76
attack	2000	0.5	0.004

$EQ_R = 0.0$
$EQ_G = 1.0$
noise gain $= 0.02$
impulse value $= 1.0$

Dynamic Model

Figure 2.19 shows a simple model for blown instruments, implemented using UPEs. In Chapter 1, *Modeling Musical Instruments*, Section *Computational Model for Organ Pipes and Flutes* I developed a computational model for organ pipes, flutes, and recorders based on their physical behavior. The basic observation used to develop the model was that a blown musical instrument can be viewed as a nonlinear forcing function at the mouthpiece, exciting the modes of a linear tube. In this section, I present an implementation of the model using UPEs.

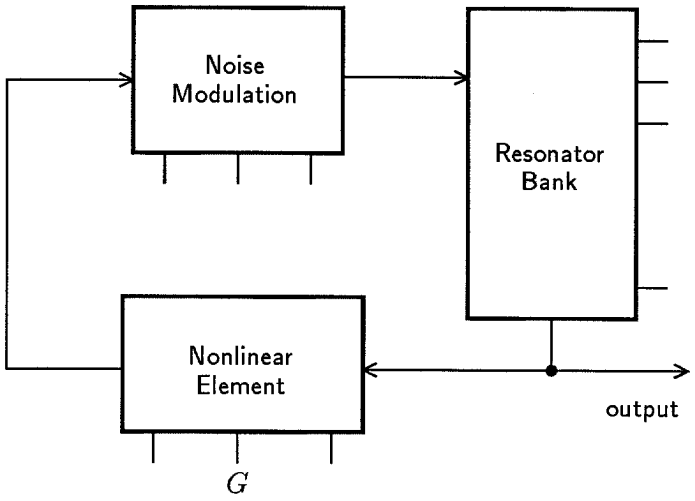

Figure 2.19 Dynamic model used for blown instruments.

We found that the computational model is composed of three pieces: (1) a linear element representing the flute body or pipe, (2) a nonlinear element representing the interaction of the air jet with the pipe, and (3) a noise-modulation section. The linear and nonlinear elements are shown explicitly in Figure 1.6; the noise-modulation section is implicit in the mouthpiece. We can translate the model into a form directly solvable on a system of UPEs by interpreting the model in Figure 1.6. The mouthpiece section serves two functions: (1) it terminates the body at the blowing end by reflecting incoming waves back into the body, and (2) it supplies energy to complement the acoustical vibrations within the body. Considering only the first function of the mouthpiece, the system is a pipe open at both ends. Such a system has normal modes of vibration, the frequencies of which are proportional to the length of the pipe and the relative amplitude and damping of which are dependent on the material composing the pipe and the width of the pipe. As we did in developing the struck-instrument model, we model the normal modes of the tube explicitly, using digital resonators.

The function at the mouthpiece, not including reflection of the incoming wave, is a hyperbolic-tangent function relating the outgoing wave to the incoming wave. For a limited range of input values, a cubic polynomial of the form presented earlier is a good approximation to a tanh, as seen in Figure 2.20.

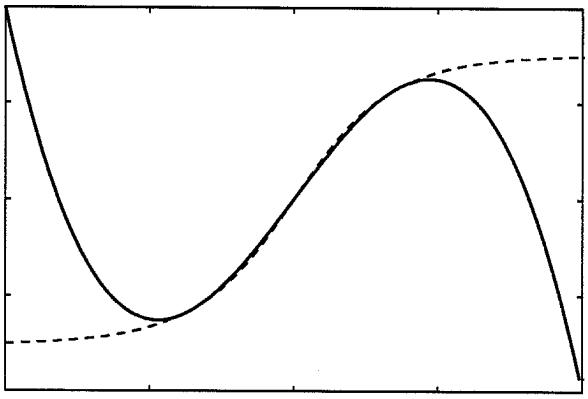

Figure 2.20 Comparison of tanh and cubic polynomial.

The noise-modulation scheme is the same as in the struck-instrument model: Noise is added to the signal in an amount proportional to the amplitude of the signal.

In summary, the UPE dynamic model is composed of three pieces: (1) the nonlinear element that computes a third-order polynomial; (2) the noise-modulation section, which adds an amount of noise proportional to the size of the signal at its input; and (3) the resonator bank, which has second-order resonators tuned to frequencies corresponding to the resonances of the pipe.

These elements are connected in a cascade arrangement, forming a closed loop. When the closed loop gain is sufficiently high, and the system is disturbed, the system oscillates with modes governed by the tuning of the resonator bank. Typically, the gain of the loop is controlled by the gain of the nonlinear element G. For small values of G, the feedback is too small and the system does not oscillate. If G is just large enough, the system oscillates with a pure tone as it operates in the nearly linear range of the nonlinear element. If the nonlinear gain G is set to an even higher value, the signal is increased in amplitude and is forced into the nonlinear region. The nonlinearity shifts some energy into higher frequencies, generating a harsher, louder tone.

In a typical application, the loop gain is set by controlling the nonlinear gain G according to the velocity of a keypress on a pianolike keyboard. A slowly pressed key corresponds to a small value for G and thus generates a soft pure tone. A quickly pressed key corresponds to a larger value for G, and hence to a louder, harsher tone. When the key is released, G is returned to some small value—one that is just under the point where the loop gain is large enough to sustain oscillation. Because G is not returned to zero, the signal dies out exponentially with time, with a time constant that is controlled by the value of G used.

A small amount of noise is injected constantly into the loop, using the noise-modulation section, so that the system will oscillate without an impulse being sent to excite it.

This model has been used successfully for generating flutelike tones. It works surprisingly well, considering that the tanh function is only approximated with a

cubic polynomial. The essence of the physical sound seems to be captured by the combination of a nonlinearity in a feedback loop with a linear element.

In physical organ pipes, the nonlinear function is nonsymmetric—it is offset and does not pass through the origin. The same effect can be achieved in our polynomial function, and we have had great success generating sounds of various timbres.

In physical organ pipes, it has been observed that only the fundamental frequency and a small number of harmonics survive the interaction of the jet with the air column [FLETCHER 80]. This observation implies that, for high-frequency harmonics, the pipe acts as a passive radiator, suggesting a modification to our model: Not all the resonator outputs are summed and fed back to the nonlinear element, but instead a subset is summed independently and behaves passively. Experiments with this idea have yielded encouraging results. The system was much more stable and controllable, producing a wider range of timbres than could be achieved before the modification.

How do we control the frequency of oscillation? Let us consider the question for a simplified version of our dynamic model. Intuitively, we may think that the system should oscillate at the center frequency of the resonator. The intuition is not quite correct. We will consider the case where the linear element is composed of only one resonator and there is no noise modulation. Further, we will consider only small-scale (low-amplitude) oscillations. From the Barkhausen criterion we know that the system will oscillate at a frequency such that the phase around the loop is a multiple of 2π radians and the loop gain is greater than unity. The phase around the loop is attributable to the sum of the delay through each section:

$$\Phi_{loop} = \Phi_{fixed} + \Phi_{res} + \Phi_{nle},$$

where Φ_{fixed} is the phase due to the fixed delay in samples, including the pure delay through all of the elements in the loop. This delay is fixed in samples but varies in radians with frequency such that

$$\Phi_{fixed} = -N \cdot \theta,$$

where N is the number of samples, and $\theta = 2\pi f/f_s$ in radians per sample. Φ_{res} is the phase through the second-order resonator and is a function of Q, f_c, and f, as shown in the Section *The Digital Resonator*. Φ_{nle} is the phase through the nonlinear element and is π radians for $G > 0$, because the output of the nonlinear element is negative for positive input and is zero for $G < 0$. Figure 2.21 shows a schematic representation of the system with the proper phase delays and a block with a linear gain of K, allowing positive K only, combining the various points of linear gain in the loop. The nonlinear gain is denoted as G.

The nonlinear function in this case has only a cubic (symmetric) component. Because the only degree of freedom in the system with respect to phase is in the resonator, it is clear that the system will find a frequency along the resonance curve for the resonator such that the phase around the loop is 2π radians, assuming that there exists sufficient loop gain. Given Q, f_c, and the sign of G, it is possible to find the frequency of oscillation by finding the point on the phase response for the resonator where $\Phi_{loop} = 0$. Similarly, the minimum value for the product GK

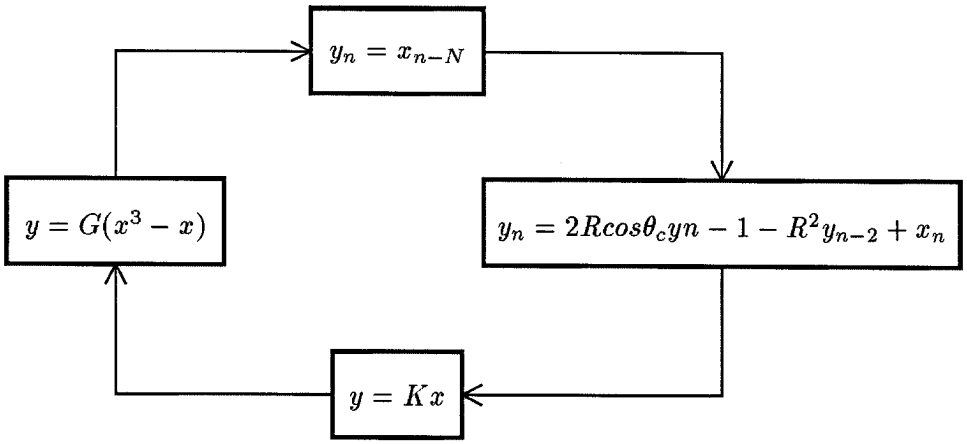

Figure 2.21 Nonlinear oscillator.

can be calculated by finding the associated gain through the resonator, given the oscillation frequency, then adjusting GK so that the loop gain is unity. Here we will simply check the expected results against measurements of the working system by calculating the loop gain and phase at a measured frequency.
Resonator parameters:

$$\begin{aligned} Q &= 5 \\ f_c &= 440 \\ f_s &= 44057\,. \end{aligned}$$

Gain parameters at minimun amplitude oscillation:

$$\begin{aligned} G &= 0.5 \\ K &= 0.00056 \\ GK &= 0.00028\,. \end{aligned}$$

Calculated resonator gain and phase at measured frequency:

$$\begin{aligned} \Phi_{res} &= -1.929\,rad = -100.52^\circ \\ \text{Gain} &= 3592.00\,. \end{aligned}$$

Fixed delay:

$$\Phi_{fixed} = 21 \cdot 2\pi \cdot \frac{446.56}{44057} = 1.3374\,rad = -76.63^\circ\,.$$

Calculated loop gain and phase:

$$\begin{aligned} \Phi_{loop} &= 180^\circ - 110.52^\circ - 76.63^\circ = -7.15^\circ \\ \text{Gain} &= 3592.0 \cdot 0.00028 = 1.0058\,. \end{aligned}$$

The phase delay is within 2% of that expected, and the gain is within 1%.

Composite Model

Because both models contain several parts in common, they can be combined into one structure, with the addition of an extra coefficient to control the feedback, as shown in Figure 2.22. A detailed view of the composite model is shown in the form of a computation graph for our computing engine in Figure 2.23. Each rectangle in the graph represents the operation of add–multiply–delay and, optionally, mod 2^{32}.

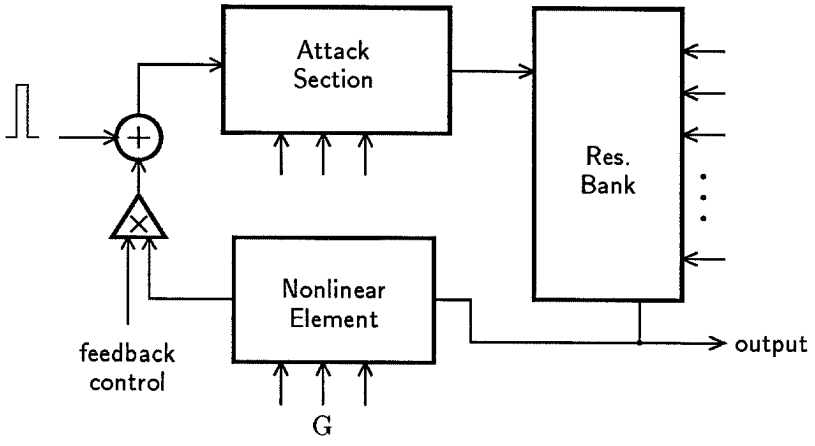

Figure 2.22 Composite instrument model.

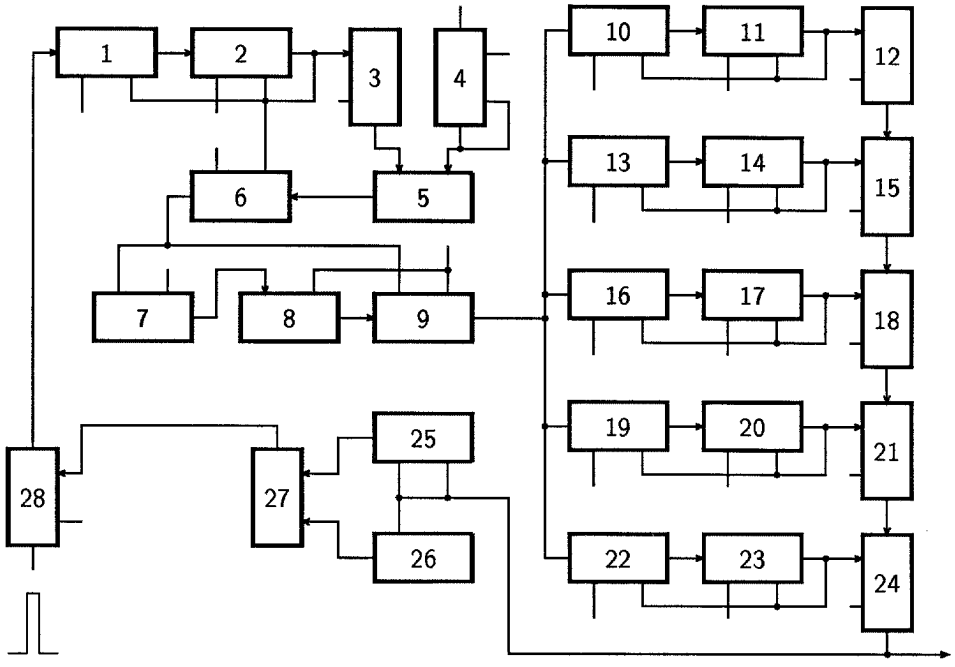

Figure 2.23 Computation graph for composite model.

Chapter 3

A VLSI Architecture

In this chapter I present a computer architecture designed specifically for the finite-difference computations used in generating musical sounds. Our computational task is the real-time evaluation of the fixed computation graphs of the variety presented in the previous chapter. In these graphs, each computation node is one or more members of the set of operations: plus, times, mod 2^{32}, and delay. Input to each node is either the output of another node or an externally supplied coefficient. Samples and coefficients flow to computation nodes across the arcs of the graph. Each processor in our computer is mapped to one and only one node and each communication channel in our machine is mapped to one and only one arc. This concept of a one-to-one mapping is a deviation from the traditional approach, in which a single processor is time-multiplexed to perform the function of each node sequentially, and memory is used to form "interconnect."

Architectural Overview

Our machine is structured as a number of inner-communicating chips, each responsible for computing a piece of a computation graph. We assume that our task may be organized such that somewhat independent subgraphs may be split off and solved fairly independently in a small number of clustered chips (possibly just one), alleviating the need for very high bandwidth between chips and between clusters of chips. Musical sound synthesis has this locality property.

Figure 3.1 shows a typical system configuration (many others are possible). The chips are organized in a ring structure with each chip communicating to its nearest neighbors. They are controlled by a global master, or *host*, that provides initialization information and coefficient updates during the computation. The host also provides an interface either to an external controlling device, such as a pianolike keyboard, or to a disk file containing musical-score information.

Each chip comprises three major pieces, as illustrated in Figure 3.2. An array of identical processing elements responsible for arithmetic and delay operations forms the first piece. The second piece is a buffer for holding coefficients supplied by the host; these coefficients, with the outputs of other processing elements, serve

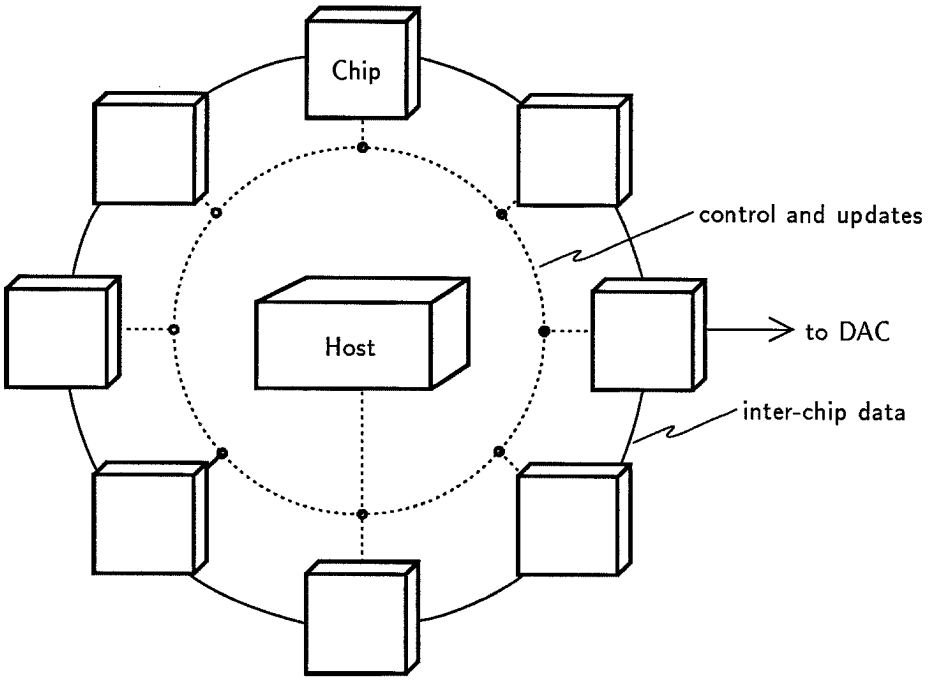

Figure 3.1 Typical system configuration.

as operands for the processing elements. The third piece is an reconfigurable interconnection matrix that serves all the chips communication needs; it connects processing elements to one another, to the output of the coefficient update buffer, and to input and output connectors of the chip. The exact patterns of communication are determined by setting switches in the matrix prior to the computation. Throughout a computation, these switches remain constant, and thus the topology of the computation graph is fixed; topology can be changed, however, between computations.

Quantization Errors and Number Representation

Errors exist in our system due to the quantization of numbers used to represent signals and coefficients. Of course, it is the goal of every system designer to minimize these errors. We needed to select a number representation (fixed versus floating, number of bits) and a method of arithmetic that would reduce the effects of quantization to a tolerable level.

Quantization of Coefficients. In signal processing systems, using a finite number of bits for the representation of coefficients results in imprecise pole and zero placement. In essence, coefficient quantization causes a system to exhibit a *finite* set of behaviors. A problem arises because the behaviors are not evenly distributed. Consider the second-order difference equation we used as a resonator:

$$y_n = 2R\cos\theta_c y_{n-1} - R^2 y_{n-2} + x_n.$$

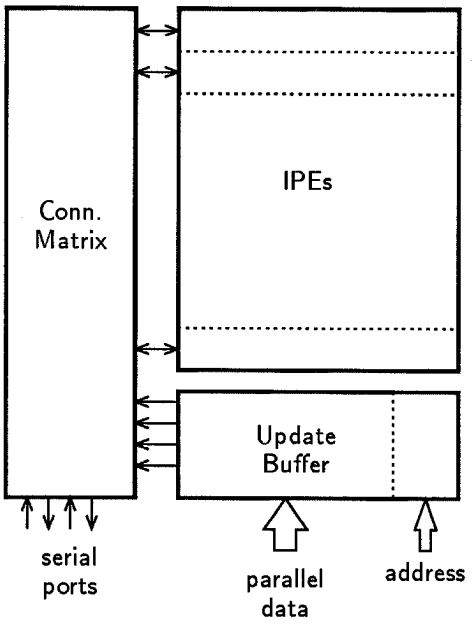

Figure 3.2 Chip organization.

This system has a pair of poles that appear in the z-plane at $z = Re^{j\theta_c}$ and $z = Re^{-j\theta_c}$. The system impulse response is a damped sine wave with frequency and damping related to θ and R. With $R = 1$, the impulse response is a sine wave of constant amplitude; the system is an oscillator. The coefficients R^2 and $2R\cos\theta_c$ may take on only a finite set of values; therefore, the poles may take on only a finite set of positions in the z-plane. In a musical context this means that only a particular set of frequencies and damping values are attainable. At the extremes of frequency and damping, the attainable frequencies are very sparse. To make matters worse, pitch is perceived as the log of frequency, further spreading out the attainable frequencies in the low range. A small number of bits are not enough to generate a frequency range with enough resolution to be musically useful. In fact, human frequency discrimination has been measured at about 0.2 or 0.3% [PICKLES 82]. To satisfy this tolerance for the fundamental frequencies of a piano, generated using the second-order resonator as a test case, at least 24-bit fractional coefficients are necessary.

Of possibly greater concern is *relative* frequency. Two tones that are meant to have an exact ratio in their frequencies may generate beat frequencies because of errors in the ratio due to quantization of coefficients.

From the second-order difference equation we have calculated the number of bits of coefficient needed to satisfy accuracy constraints on the center frequency. The heavy curve shows the number of bits of coefficient needed to satisfy a tolerance of 0.2% frequency discrimination. This tolerance is important for *absolute* frequency discrimination. The lighter curve shows the number of bits needed to get within 1 Hz of a given desired frequency. This curve is important for *relative* frequencies, to maintain a minimum beat frequency of 1 Hz. The curves were generated by

effectively sampling the frequency (horizontal) axis and then incrementally increasing the number of bits in the coefficient until the resulting center frequency for the resonator came within the tolerance. The number of bits to satisfy a given tolerance at any one *particular* frequency may be very small but in general the number could be relatively large and is shown by the "staircase" shaped curves. Of course, the number of bits, in both cases, goes up as the tolerances are made smaller. Also, the curves move up slightly for smaller values of R.

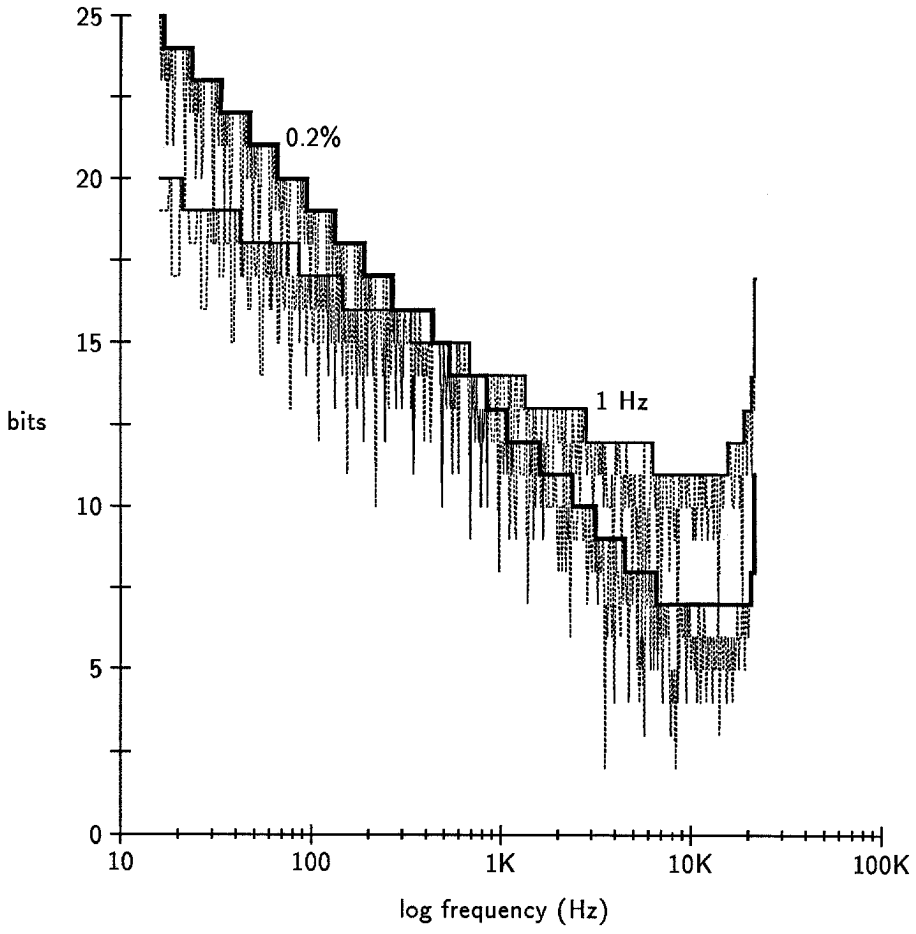

Figure 3.3 Effect of coefficient size on center frequency accuracy. The curves shown are for a 2-pole second-order system with $R = 1.0$. The heavy curve shows the number of bits of coefficient needed to satisfy a frequency tolerance of 0.2%. The lighter curve shows the same information for a frequency tolerance of 1 Hz.

Quantization Noise. Each time a digital multiplication is performed on two N bit numbers, a $2N$ bit result is formed, and the result must be rounded (or truncated)

to fit within the N bit number representation. Rounding introduces a small error signal (less than one LSB). Some researchers have approximated this effect by modeling it as a noise signal added to the signal at each multiplier in a system [OPPENHEIM 75]. Although this model assumes that the error signal is a white-noise sequence, is uniformly distributed, and is uncorrelated with the signal, it is probably fairly accurate for signals as complex as those in music. Using this model, the effect of quantization may be derived for various systems or filter forms. Oppenheim and Schafer computed the variance of the output noise due to arithmetic rounding in a second-order two-pole system. Using their result we have computed the output noise for several values of center frequency and R, as shown in Figure 3.4. The values of R were chosen to cover a wide range of values for the damping constant τ. The sampling frequency is the digital audio standard of 44057 samples/second. The output noise level was computed in terms of the *number of bits* necessary to represent such a level. From Figure 3.4 it is apparent that, for systems with long time constants, at least 16 bits of each sample may be in error due to quantization; therefore, a number representation with much more than 16 bits is necessary.

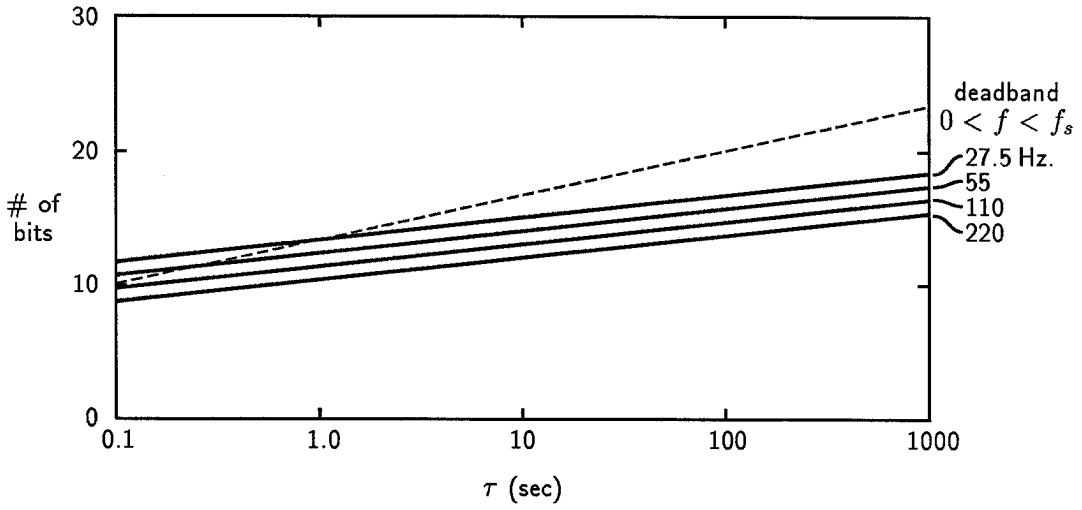

Figure 3.4 Quantization effects. The solid lines show the amplitude of noise introduced into a two-pole system because of quantization, for several values of center frequency. The dashed line shows the amplitude of limit cycles introduced because of quantization of state values.

Limit Cycles. One type of limit cycle in digital systems results in *deadbands*, or intervals of signal values around zero in which a system can experience self-sustaining oscillations because of rounding of state values [JACKSON 69]. As before, we computed the effect in the *number of bits* for a second-order two-pole system. The amplitude of the limit cycle in the second-order system is only a function of the damping coefficient R. This amplitude (in bits) is plotted, with the curves showing

the quantization noise in Figure 3.4. Limit cycles are potentially a more harmful problem than is quantization noise because limit cycles concentrate their energy at a particular frequency unlike quantization noise that is more uniformly distributed. These curves may be interpreted as bounds on the quantization noise and limit-cycle amplitude and used as a guide for choosing the number of bits to use to represent signals. From the curves, 16 or even 24 bits clearly are not enough. We have chosen a two's-complement representation that employs a sign bit, two integer bits, and 29 bits of fraction.

This analysis says nothing about composition of second-order sections or even other filter forms, but does treat an important case and gives a flavor for how bad quantization effects can be in general.

The Processors

Bit serial processing offers two attractive features for our application. First, the processing elements are physically small, so large numbers of them can be integrated on a single chip. Bit serial processing also facilitates bit serial communication, simplifying communication channels; single wires can be used to interconnect processing elements. One potential drawback is the *latency* incurred with each operation—the time from the operand's arrival until the total answer's arrival at the output. In our application, however, we *want* a delay at each processing step, so the latency is an advantage.

Various bit serial multiplication schemes have been implemented and presented in the literature [LYON 76]. We wished to provide maximum processing power per unit chip area as was possible with current technology. Therefore, we chose the simplest multiplication scheme that met the constraints placed by standard digital audio rates. Our processor is a serial–parallel multiplier structure capable of one multiply–add–delay step per word time; we call it an *inner product element* (IPE). The multiplier structure is simple and therefore requires little space to implement in silicon [MEAD 85]. Inputs arrive one bit at a time, LSB first, and the output is generated one bit at a time. All inputs and outputs have the same number representation; therefore, there are no restrictions for interconnection of processors or the connection of coefficients. The processing element is described in detail in the section *Processing Element.*

The Connection Matrix

The connection matrix provides point-to-point communication between processors, from the update buffer and bidirectionally with the outside world. The matrix is *programmable*; the interconnection patterns within the matrix are not fixed but are changeable from external control, made possible by a storage cell located at each cross point in the matrix and circuitry to set the state of the storage cells. In addition to programmability, the connection matrix also takes advantage of the inherent locality in sound synthesis computations and is discretionary in the allowable interconnection patterns, saving in the chip area and providing for growth of the processing power of VLSI implementations.

Figure 3.5 shows the basic structure of the connection matrix and its interface to the other components. Note that the horizontal wires, or *tracks*, are used to bring in signals from off-chip and to send signals off-chip, as well as to provide communication

between processing elements. One possible configuration is to dedicate one track-per-processing element output, which guarantees that any IPE can communicate with any other IPE. Such a configuration, however, grows as the square of the number of processing elements; in musical sound synthesis applications, it is a waste of chip area. In Figure 3.6, we have mapped the computation graph in Figure 5 onto the processor array by assigning nodes of the graph to IPEs and routing the interconnections, assuming that tracks could be broken arbitrarily. The assignment of processors to nodes in the graph was ordered from left to right across the array for consecutive number nodes. Clearly, all tracks have many breaks and there are a large number of small links and a relatively smaller number of larger links, and so on. In the modification shown in Figure 3.7, tracks no longer span the entire array of IPEs, but rather are split at one or more points along their lengths. There is one track of links for length 2, one for length 4, and so on, doubling the length of the links for each track until the entire length of the array is spanned in one link. The breaks in the tracks are arranged to avoid any two breaks lining up vertically, and consequently to maximize the potential communication between pairs of processors. This matrix grows as $N\log N$—rather than N^2, as does the earlier version—thus saving area. The computation graph of Figure 2.23 has been mapped into the new structure, in Figure 3.8. The ordering of the nodes in the graph has been perturbed to make a better match. All but one network is routed in the modified matrix; one additional track is used to handle that network.

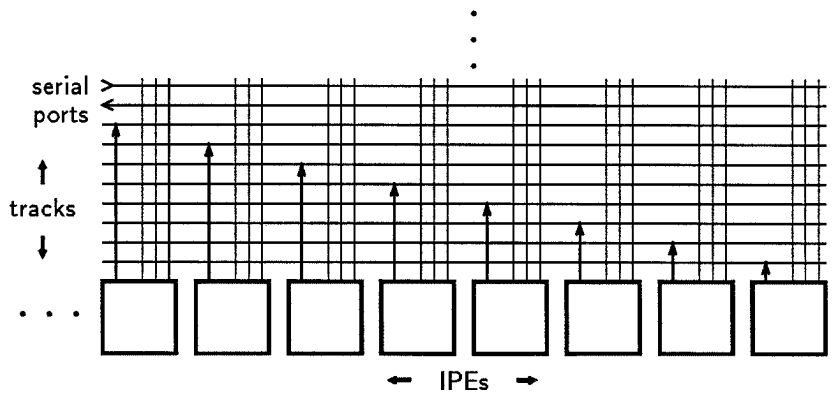

Figure 3.5 Basic structure of connection matrix.

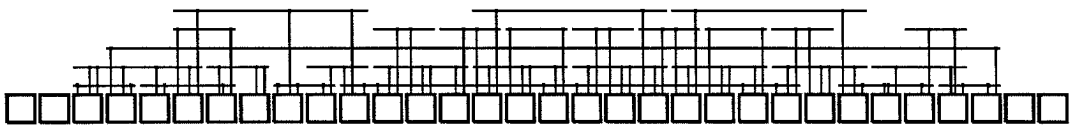

Figure 3.6 Mapping of computation graph to processor array.

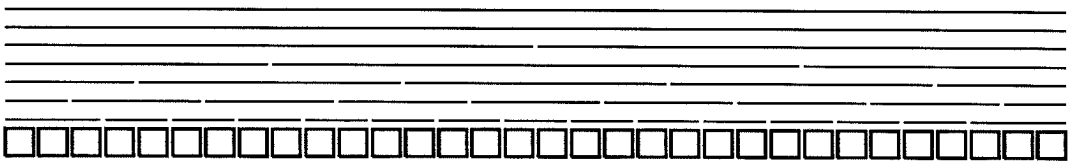

Figure 3.7 Discretionary interconnect matrix.

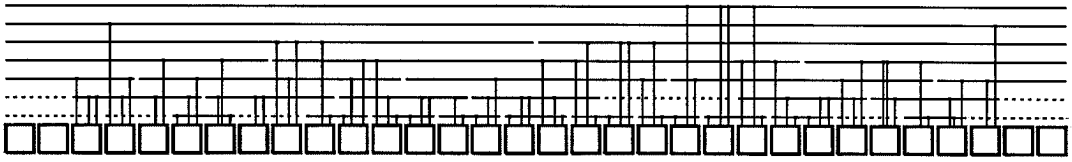

Figure 3.8 Mapping of computation graph to modified matrix.

It is not known what the optimum configuration for the connection matrix is, and what are the best algorithms for assignment of computational nodes to processors in the array. Although the assignment problem is NP-complete, heuristic algorithms that find the inherent localities in our applications perform very well and are aided through the addition of a few extra tracks in the matrix and few extra processing elements in the array.

The Update Buffer

The update buffer is simply a register bank to hold coefficients (that is, inputs to the processing elements supplied from the host computer). Input to the update buffer is a parallel connection to a standard computer memory bus. The outputs of the update buffer are bit serial lines that run through the connection matrix to the processing elements. For maximum flexibility in the assignment of processing elements, no *a priori* correspondence is made between update buffer registers and processing elements; this assignment is made by programming the connection matrix.

One important feature of the update buffer is that it is double buffered. Coefficients can be sent from the host computer to the update buffer without affecting the ongoing computation. Only after all the coefficients of a new set of updates have arrived in the buffer, is a signal sent to update them simultaneously. If some coefficients were allowed to change before others did, instability could result.

Empirically, we have found that our applications average about one coefficient per processing element. This fact defines the nominal number of registers in the update buffer to be the same as the number of processing elements, with a few spares to cover exceptional cases.

The structure of the update buffer comprises two RAM structures laid one on top of the other (Figure 3.9). The first RAM is writable from the parallel input bus with a decoder that selects one coefficient (row). The second RAM is readable one bit (column) at a time, all coefficients being read simultaneously. A select signal cycles through the columns of the second RAM one bit at a time, sending the bits

of the registers to the output, LSB first. Under control from the host computer, a transfer signal copies the contents of the first RAM to the second one.

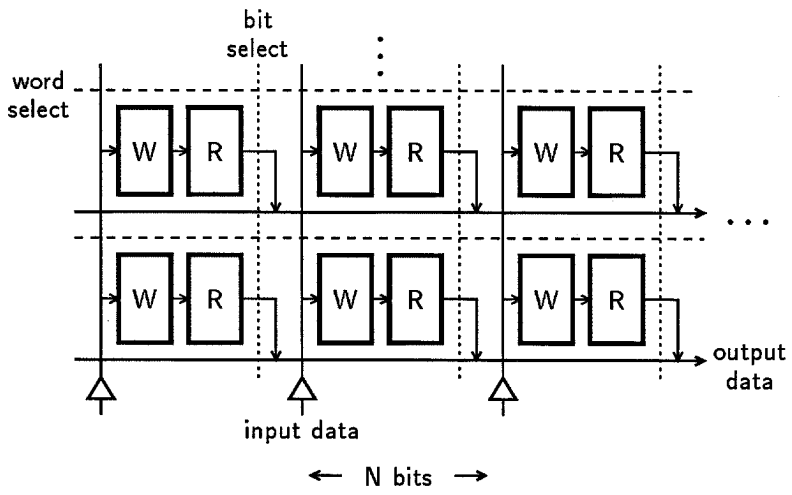

Figure 3.9 Dual ram structure of update buffer.

Processing Element

The structure and function of the IPE is apparent from examining long-hand multiplication.

$$
\begin{array}{cccccccc}
 & & & b_3 & b_2 & b_1 & b_0 & B \\
 & & \times & m_3 & m_2 & m_1 & m_0 & M \\
\hline
 & & & m_0b_3 & m_0b_2 & m_0b_1 & m_0b_0 & \\
 & & m_1b_3 & m_1b_2 & m_1b_1 & m_1b_0 & & \\
 & m_2b_3 & m_2b_2 & m_2b_1 & m_2b_0 & & & \\
m_3b_3 & m_3b_2 & m_3b_1 & m_3b_0 & & & & \\
\hline
y_6 & y_5 & y_4 & y_3 & y_2 & y_1 & y_0 & Y
\end{array}
$$

Two operations are being performed—every bit of the multiplier M is multiplied by every bit of the multiplicand B, and the resulting partial products are summed together.

Many architectures exist for performing multiplication, covering the full range of tradeoffs between the number of components (*area* in VLSI implementations) and speed. A simple view of the tradeoff is seen by contrasting a single serial full-adder scheme to a parallel multiplier. The single full-adder scheme has a *minimum of components* (one full-adder and one carry flip-flop) but requires n^2 operation steps. At the other end of the tradeoff space are parallel multipliers; they generate all

n^2 one-bit products simultaneously and thus minimize the *time* needed to perform the multiplication at the cost of n^2 components.

The other major issue in a parallel computer architecture such as ours, is the form of the data communicated between processors. In a VLSI implementation, when a large number of processors are interconnected, anything other than single wire interconnections is prohibitive. The form of communication does not necessarily need to correspond to the number representation for processing, but conversion between representations increases system complexity.

In sound synthesis, a bit-serial scheme fits naturally in the appropriate place in the area–speed tradeoff space and allows easy communication between processors. It uses $O(N)$ full-adders and operates in $O(N)$ time. Many schemes fit into this category. Lyon developed some "bit-serial pipelined" multipliers that function in N-bit times and work with two's-complement numbers [Lyon 76]. He also developed a scheme to use a modified Booth encoding to reduce the number of stages required. Our objective was to develop a processor with minimal area but that met the rather modest speed requirement of digital audio synthesis (50 K samples/second).

Serial–Parallel Multiplier

A scheme that uses N full-adders is shown in Figure 3.10. It is commonly referred to as a serial–parallel multiplier. The multiplicand B is supplied in parallel form and the multiplier M is broadcast bit-serially to all stages. Each stage contains a full-adder, a carry flip-flop, and a delay element for passing values from stage to stage. For $N = 4$, the multiplication procedure begins by broadcasting m_0 and generating the partial product (m_0b_3 m_0b_2 m_0b_1 m_0b_0). Next, m_1 is broadcast to generate (m_1b_3 m_1b_2 m_1b_1 m_1b_0). Simultaneously, the last partial product generated is shifted one stage to the right and summed with both the new partial product and the carry bits from the last operation. After m_0 through m_3 have been broadcast, the result bits y_0 through y_3 have been generated. A sequence of four zeros is broadcast and the high four bits of the result are generated. The entire multiplication operation requires 8 cycles to multiply two 4-bit numbers and in general requires $2N$ cycles to multiply two N-bit numbers.

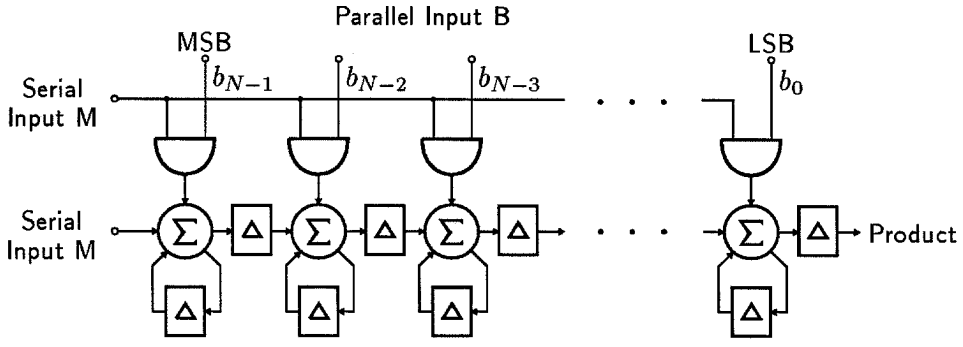

Figure 3.10 Serial–parallel multiplier structure.

During the multiplication, at every step, each stage receives a bit from the stage immediately to its left and adds the bit to the one-bit product and the previous carry. Stage $N-1$, however, has no stage to its left. The input to stage $N-1$ is supplied externally from input A. The input need not be 0, in which case the input A are added to the high order N-bits of the result Y.

Serial Input for B

The first modification to the multiplication structure allows B to be shifted into the array serially. A holding register and a shift register as in Figure 3.11 is added to the multiplication structure. The multiplicand B enters the shift register LSB first and is shifted until the MSB is about to enter; then the LDB signal transfers B to the register used in the multiplication. Note that B must enter the array one word cycle before M and A to ensure its existence for the LSB of M.

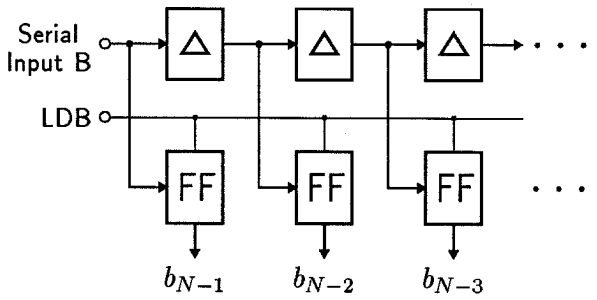

Figure 3.11 Shift register and holding register for B.

Extension to Two's Complement

The scheme described above works correctly for positive numbers only. We have adopted a two's complement number representation and extended the basic serial–parallel multiplier structure to handle two's complement numbers correctly.

The first extension allows the multiplier M to be negative. As already noted, the N-bits of M are used to generate the result bits y_0 through y_3, after which four more bits, presented as 0, are broadcast to the array. Correct operation for two's complement numbers results from sign extending M to $2N$ bits and broadcasting all $2N$ bits. Note that for positive numbers, zero is the sign extension.

The second extension allows the multiplicand B to be negative and is more complicated. All two's complement numbers are represented as:

$$B = \sum_{i-0}^{N-2} b_i \cdot 2^i - b_{N-1} \cdot 2^{N-1}.$$

Note that the MSB of B, b_{N-1}, carries negative weight, meaning that the $N-1^{\text{st}}$ stage in the array must subtract its product $m_i b_{N-1}$, whereas all other stages add their product. Therefore the full-adder and carry bit of stage $N-1$ become a

full-subtractor and borrow bit. A full-subtractor is formed from a full-adder by inverting the subtrahend input to form the augend for the full-adder and inverting the sum to form the difference. This technique for modifying a full-adder to form a full-subtractor is particularly simple when dual-rail logic is used, since inversion is achieved simply by exchanging rails. The resulting modification in stage $N-1$ is shown in Figure 3.12.

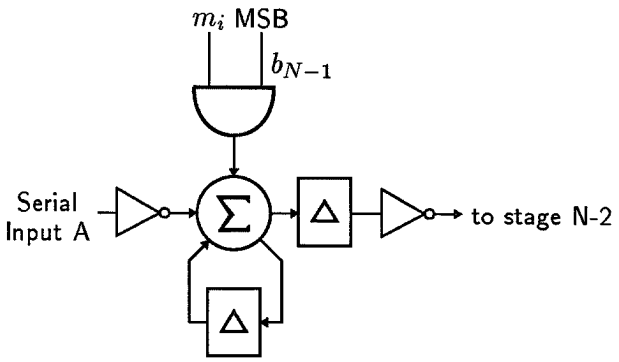

Figure 3.12 Modified stage 1 for two's complement.

Standard Fixed Point Number Representation

One goal in the design of our system is that one single number type is sent both from processor to processor and to and from the outside world. It is simplest to use an integer-based number representation from the point of view of a hardware designer. However, the burden is shifted to the user, who must assure that scaling (shifting) operations are performed periodically to keep numbers within range. We have adopted a fixed-point number representation composed of a sign-bit, 2 integer bits, and 61 bits of fraction: $sxx.xxx\cdots x$. The word cycle for the IPE is 64 bits, and a 64-bit result is formed, although for simplicity only 32 bits of input are used. Usually the low-order 32 bits are ignored. Several shift registers acting as delays are needed to align inputs to the multiplier structure properly.

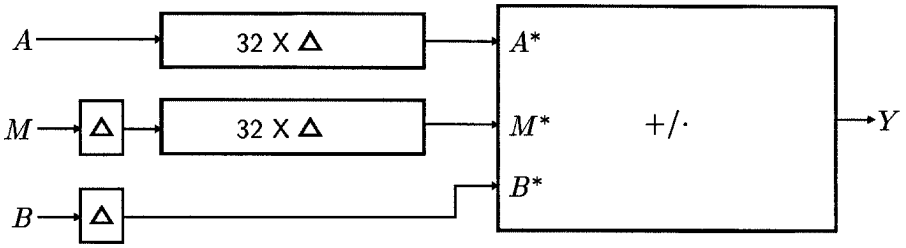

Figure 3.13 IPE with appropriate delays on inputs.

Figure 3.14 shows the timing diagram for one complete IPE operation. During the first half of the word cycle, the high-order output bits from the previous operation leave the array, B enters the array and is transferred to the holding register, 0 is entered at the A input, and M is sign-extended for the previous operation. During the second half of the word cycle, B is ignored, M is broadcast to the array, the bits of A enter the left-most stage and are summed ultimately with the high bits of Y, and the low bits of the result Y leave the array. In normal operation, the timing diagram of one IPE is abutted with that of another so that the high bits of Y leave one IPE as the other requires input.

The delay from the time that the inputs enter the IPE until the high 32 bits of the result leave the IPE is 64 bit periods—one word-time delay. Therefore, the function of the basic IPE is expressed as: $Y = (A + BM)z^{-1}$.

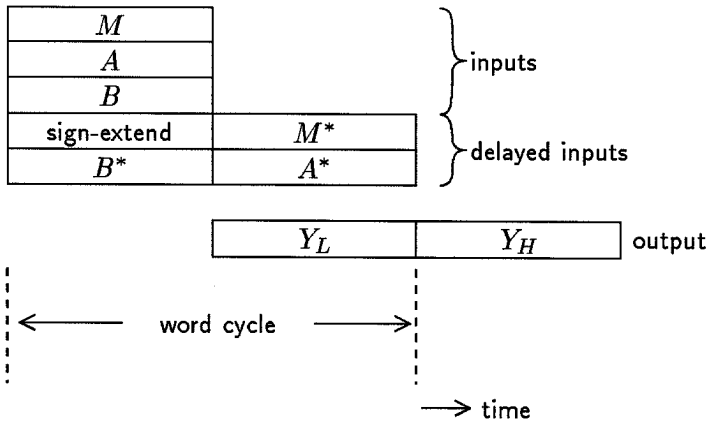

Figure 3.14 Input and output timing for the IPE.

Computing the Mod Operation

The IPE generates 64-bit results, but usually only the top 32 bits are used as inputs to other IPEs; however, the low bits of the result are not always discarded. Taking the low n bits of a number with greater than n bits results in mod 2^n. The mod operation is useful in sound synthesis applications. For example, the expression $y_n = p \cdot \bmod 2^n + q$ generates random samples for the proper choice of p and q [KNUTH 68].

In our system, if the signal that loads B into its holding register comes one-half word-time earlier, then the low bits will be loaded into the holding register rather than the high bits and the result will be mod 2^{32}. A bit of state is supplied from the connection matrix to each IPE and is used to determine the time when the load B (LDB) signal will occur, and consequently if B or $B \bmod 2^{32}$ is loaded.

Linear Interpolation

Linear interpolation is a function that is useful in musical sound production for a variety of purposes, including the mixing of sound streams, implementing ladder

filters, and approximating delays and waveforms from tables. Any multiplication scheme can be modified to perform linear interpolation with very little increase in complexity. We will demonstrate a modification to our basic serial–parallel multiplier structure to enable the computation of the linear interpolation between the input B and a new input D, using M as the constant of interpolation. The new function computed is $Y = A + MB + (1 - M)D$.

First, let us assume that M is a number in the range from 0 to +1.0—in our standard number representation: $000.m_{N-4}m_{N-3}\cdots m_0$. For this case, it is true (or we can force it to be true) that $M + 1.0 = M$. We enforce this equality by forcing any bits to the left of the binary point to zero. Using the above fact and the definition of two's complement numbers ($-M = \overline{M} + 2^{-b}$, where b = number of fractional bits), we derive an identity involving $\overline{M}$, the one's complement of M:

$$\begin{aligned} M + 1 &= M \\ M &= M - 1 \\ -M &= 1 - M \\ \overline{M} + 2^{-b} &= 1 - M \\ \overline{M} &= 1 - M - 2^{-b} \\ \overline{M} &\approx 1 - M. \end{aligned}$$

Therefore, the value $1 - M$ needed in the interpolation may be approximated by $\overline{M}$; and they differ by only one LSB. For large word size and most applications the approximation is probably close enough. It enables the computation of linear interpolations with a simple modification to our multiplication structure. At each stage in the multiplier a two input AND-gate is used to compute the product $m_j b_i$, leading to the product BM. The product $D\overline{M}$ can be generated by computing the one-bit product $\overline{m_j} d_i$ at each stage. In fact, both one-bit products can be generated simultaneously at each stage by replacing the AND-gate with a multiplexer that chooses between b and d based on m, as shown in Figure 3.15. Of course, a holding register for D is also required.

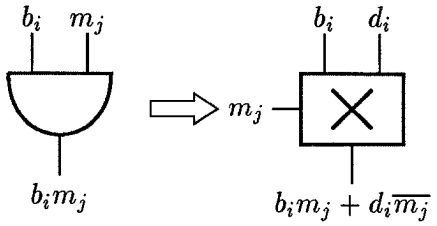

Figure 3.15 Modification to multiplier stage to enable linear interpolation.

In systems where the number representation includes nonfractional bits, to support linear interpolation the property $M + 1.0 = M$ must be maintained explicitly by a modification to the multiplexer function. The structure shown in Figure 3.16 provides the proper function by forcing the M input to both gates to zero during the time that the nonfractional bits of M are broadcast to the array of stages. The control input C is high during the time the nonfractional bits of M are being broadcast, and low otherwise.

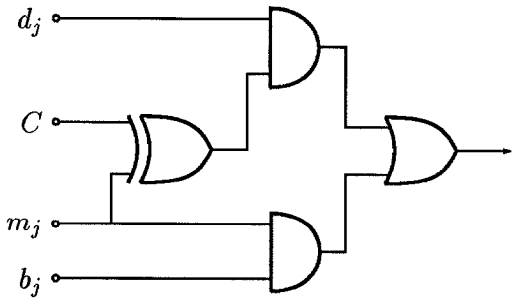

Figure 3.16 Multiplexer circuit for linear interpolation. This circuit ensures correct operation for number representations with nonfractional bits.

System Level Organization

Orchestra Model

How do we organize chips of processing elements, update buffers, and connection matrices into usable systems? Musical instrument models are structured as a number of intercommunicating chips, each chip responsible for a piece of a computation graph. We assume that the graph may be organized such that the communication between subgraphs is small, and thus the graphs may be split off and solved independently, alleviating the need for high bandwidth between chips. Sound synthesis algorithms have this locality property. Figure 3.1 shows one of many possible system configurations. The chips are organized in a ring structure with each chip communicating to its nearest neighbors. They are controlled by a global master, or *host* (conventional microprocessor system) that provides initialization information and coefficient updates during the computation. The host also provides an interface either to an external controlling device, such as pianolike keyboard, or to a disk file containing musical score information. Many other choices for system configurations are possible, for example, grid connections and binary n-cubes. Since single wires provide interchip communication, a relatively large number of inputs and outputs may be provided with each chip.

What happens as we add more instruments models to our system? The simplest way to add more instruments to our system is to add more chips to the ring

structure shown in Figure 3.1. The interchip communication lines are used to communicate results from one instrument to the other and provide some flexibility in assigning instrument topologies to processors–chips. The host now has the additional responsibility of providing coefficient updates to the additional instruments. Although instruments compute their sounds independently, their output streams sum to form composite musical sounds, therefore, communication between instruments is needed. The most practical way to achieve this is to cascade instruments in such a way that each instrument adds its output to the stream and passes it on to the next.

As the number of instruments and the number of chips grow, the ratio of the number of coefficients supplied by the host to the cycle time of the host–memory system increases to a point where the host cannot keep up in real time. This effect is quantified by examining system performance metrics. The first metric is *latency*: the time from a user action to the time that an effect is heard at the output. An obvious example is the time from the point when a note event is triggered to the time the associated sound is heard. The second measure of system performance is the *sustained note rate*: the total number of notes (for all instruments) per second. In our simple system configuration, latency is the most demanding of the two requirements and is determined by the time it takes the host to generate coefficients and transfer them to the processor chips. Let's say, for example, that our orchestra includes 100 instruments, each capable of generating one voice at a time.* A conservative maximum allowable interactive latency is 10 ms. Our orchestra is composed of "high quality" instruments, so we assume that on the average each instrument uses 100 coefficients. From these assumptions we can compute the maximum communication bandwidth needed between the host and the processor chips:

$$\frac{100 \text{ instruments} \times 100 \text{ coefficients per instrument}}{10 \text{ ms}} = 1 \text{ Million coefficients per sec,}$$

where coefficients are 32 bit numbers. This conservative communication bandwidth estimate is comparable to the processor–memory bus speed on current computers. However, this calculation accounts only for the transfer of precomputed coefficients. In practice, interactive user input will alter the values of some coefficients. If we assume that our host is capable of 4 million instructions per second (MIPS), and that each computed coefficient requires an average of 100 instructions, our host can compute about 40 thousand coefficients per second and only about 40 coefficients within one latency time! This rate is clearly insufficient for 100 instruments. Therefore, coefficient computation is likely to be the limiting factor in the growth of our system.

Both the coefficient communication bandwidth and the coefficient's computation bandwidth demands may be relieved by reorganizing the system in a way that reflects the natural structure within a real orchestra. Figure 3.17 shows such a structure. Voices are grouped together around a local host. Within a voice-group, voices work together to simulate multi-voiced and single-voiced instruments. Voices

* We make the distinction between *instrument* and *voice*. (Some instruments are multi-voiced, for instance, stringed instruments, and others are single-voiced, woodwinds, for example.)

communicate bit-serially among themselves either within a chip or between chips as before for interactions between voices of a multi-voiced instrument and simple summing of instrument outputs. The master controller is a host computer roughly equivalent to the conductor of the orchestra. The idea is that the conductor-host interprets the score information coupled with real-time user input and passes on high-level information to the appropriate voice groups. The voice-groups each interpret the high-level information in the local context of an instrument model and instrument groupings. Communication between voice-groups is necessary only for summing the outputs of the instruments. It is advantagous to group together instruments that react to common control information, sharing computation.

Within the interpreter of score information is some model of the musician as well as of the instrument. The idea of interpreting a standard score representation in the context of *player functions* and instrument models is developed by Dyer in [DYER 87]. Some user functions, such as vibrato, may even be computed in the UPE chips.

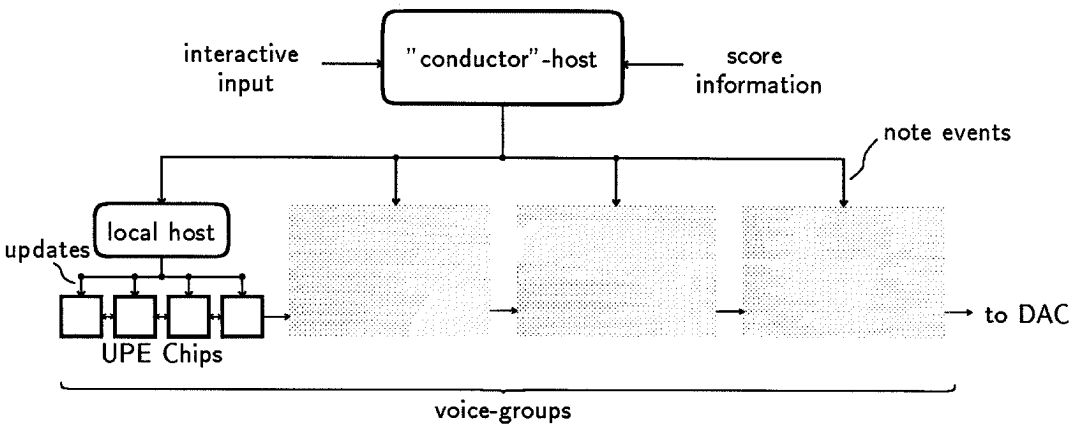

Figure 3.17 System configuration for orchestra simulation.

The amount of real-time control output a user can generate is limited and bounds the amount of control bandwidth needed between the conductor-host and the voice groups. One way to use the system is as a performance instrument, where note events for all instruments are generated on the fly. This situation is not very likely to happen in practice, however, because it is difficult to see how any one person could *play* all the instruments of an orchestra single-handedly. A much more likely situation is a mixture of live performance with pre-existing score information. We envision the system being used to develop a score interactively, layering one section of instrument scoring at a time on top of previously generated scores. This case is handled by either downloading pre-existing score information to each voice-group before the performance (if enough local memory is available), or by

using the relatively low-communication bandwidth time between note events to pre-send time-tagged note events. At performance time (runtime) the host broadcasts real-time control information derived from user interaction. The real-time control information includes tempo information (timing) and modifiers (accents, etc.) in additional to note events for instruments without a pre-existing score.

Keyboard Instruments

Keyboard instruments and pianos in particular lend themselves to a special system level organization. These instruments are typically composed of an array of strings (or set of strings) all stretched across a common bridge that is allowed to move up and down, transferring energy to the sound board. Striking a key on the keyboard excites one string and thus the bridge and soundboard, which, in turn, influences the other strings. In our system organization, each string, along with the hammer mechanism, is treated as loosely connected parallel voices. A small number of chips are used to generate each voice. A few extra chips are used to compute global coupling through the bridge and the sound board. As a consequence of all strings' sharing a common bridge and sound board, every voice in our implementation contributes to the bridge and sound board computation and receives feedback. In the case of the piano, most coefficients remain constant and need not be rewritten to the chips at each note. A global host is used to provide key-force dependent and pedal dependent coefficients to each voice, as these are the only coefficients that need to change on a per-note basis. The system may be used as a general synthesizer with one or more voices per key when not simulating a piano.

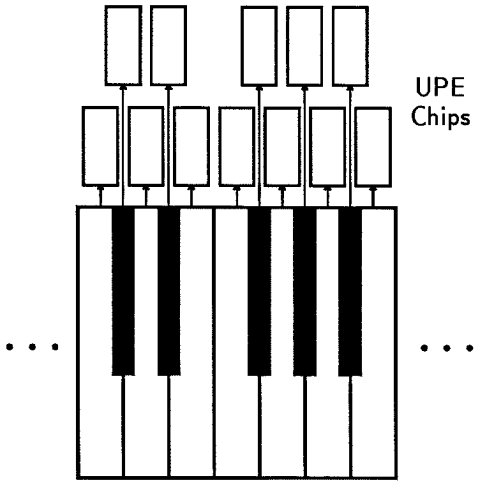

Figure 3.18 System configuration for keyboard instrument.

Chapter 4

VLSI Implementation

This chapter presents a new form of CMOS logic design and applies it to the implementation of our computing engine used in sound synthesis. Also presented are some details of an nMOS implementation of the processing element.

A number of logic forms and clocking schemes for CMOS integrated circuits are in common use. The most common logic form consists of two networks of transistors, the gates of which are connected to the input variables. An n-channel network defines the Boolean condition under which the output is connected to ground (logic zero). A p-channel network defines the complementary condition under which the output is connected to a logical one. Because, in many CMOS processes, the output of a single pass-transistor cannot be guaranteed to exceed the logic threshold of a typical inverter, either pass-transistor networks are forbidden or a complementary transmission gate employing both p- and n-channel devices is used.

Clocking schemes for CMOS currently offer tradeoffs over a wide range in risk versus efficiency space. In one scheme, a single-phase clock and its complement are distributed and are used to control either transmission gates or transistors controlling power to the p- and n-channel switching networks. Proper operation in either case requires that the logic delay of the stage exceeds the skew between the two clock lines. In a much safer approach, a two-phase clock is used, both the clock and its complement being distributed for each phase. In this case, risk is eliminated at the expense of doubling the clock wiring. Yet another form is popular in gate-level designs. A single clock is distributed and is locally inverted at master–slave storage elements. Risk in this case is eliminated at the expense of a minimum storage element employing ten or more transistors.

In this chapter, I describe a logic form that retains much of the simplicity, elegance, and compactness of the familiar two-phase nMOS form, with the added advantage of fully static operation. Formal semantics for circuits implemented in this form is easily derived without detailed circuit- or switch-level simulation.

Complementary Set–Reset Logic

A shift-register stage in complementary set–reset logic (CSRL) is shown in Figure 4.1. In this and all other examples, I use n-well technology to illustrate the principles. The circuits for p-well technology are identical, provided the p- and n-channel devices are interchanged, and the power and signal voltages are changed from positive to negative. With this convention, the same physical masks can be used to fabricate a design in either technology.

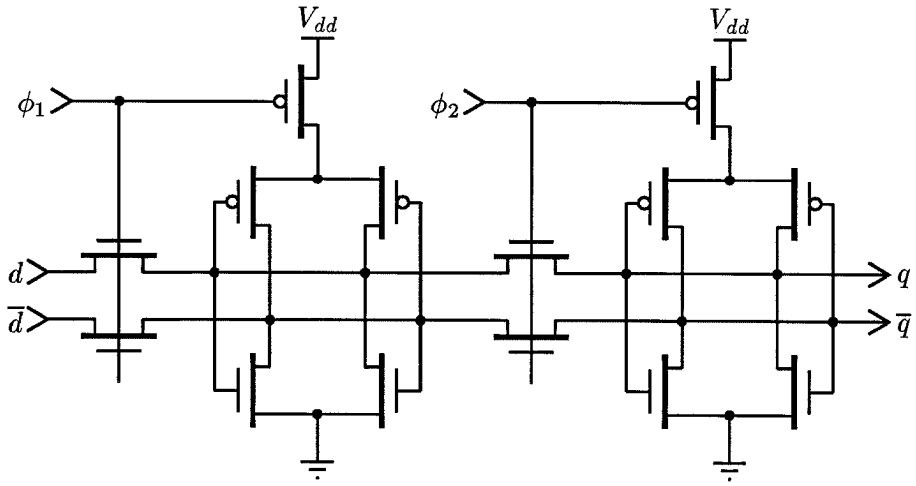

Figure 4.1 CSRL shift-register stage.

Signals are two-rail; both the data and its complement are represented. As ϕ_2 rises, the power supply to the second stage is limited by the upper p-channel power-down transistor. By the time ϕ_2 reaches the threshold of the two pass-transistors, the current to the second flip-flop is limited to about one-half the maximum that can be supplied when the clock is low. For this reason, the first stage, which is fully powered up, can force its state into the second one as ϕ_2 rises. Similarly, the second stage can transfer its contents into the following stage on the rising edge of ϕ_1. Unidirectionality of the transfer is guaranteed by the association of the power-down transistor with the receiving stage, and the mutual nonoverlap of the clocks.

The dynamics of a transfer is shown in Figure 4.2. The interesting case, illustrated in both the panels, occurs when the new data bit is different from the previously stored data bit. During the ϕ_2 period, the signal rail labeled q discharges toward ground and the $\overline{q}$ rail charges toward V_{dd}. The rates are limited by the capacitances of the nodes and the resistances of the pass-transistors. After a time t_1, the two voltages are equal. The clock ϕ_2 returns to zero at time t_2. If $t_2 > t_1$, as shown in Figure 4.2(a), a successful transfer results. There is no need to wait for the two signal rails to pass any absolute threshold. Each flip-flop can be viewed as a sense amplifier with very high differential gain. The sense-amplifier action will fully restore both data and complement as long as their *relative* values are of the correct

sign when the clock falls. Figure 4.2(b) illustrates an attempted transfer in which the falling edge of the clock preceded the time t_1 at which the two signals became equal. Under these conditions, the signals return to their previous values, and the transfer fails. The time t_1 thus represents the minimum time the clock must be high and thereby limits the maximum frequency of operation. At all times, at least one clock is low, and hence at least one flip-flop has power; therefore, the CSRL shift-register is fully static.

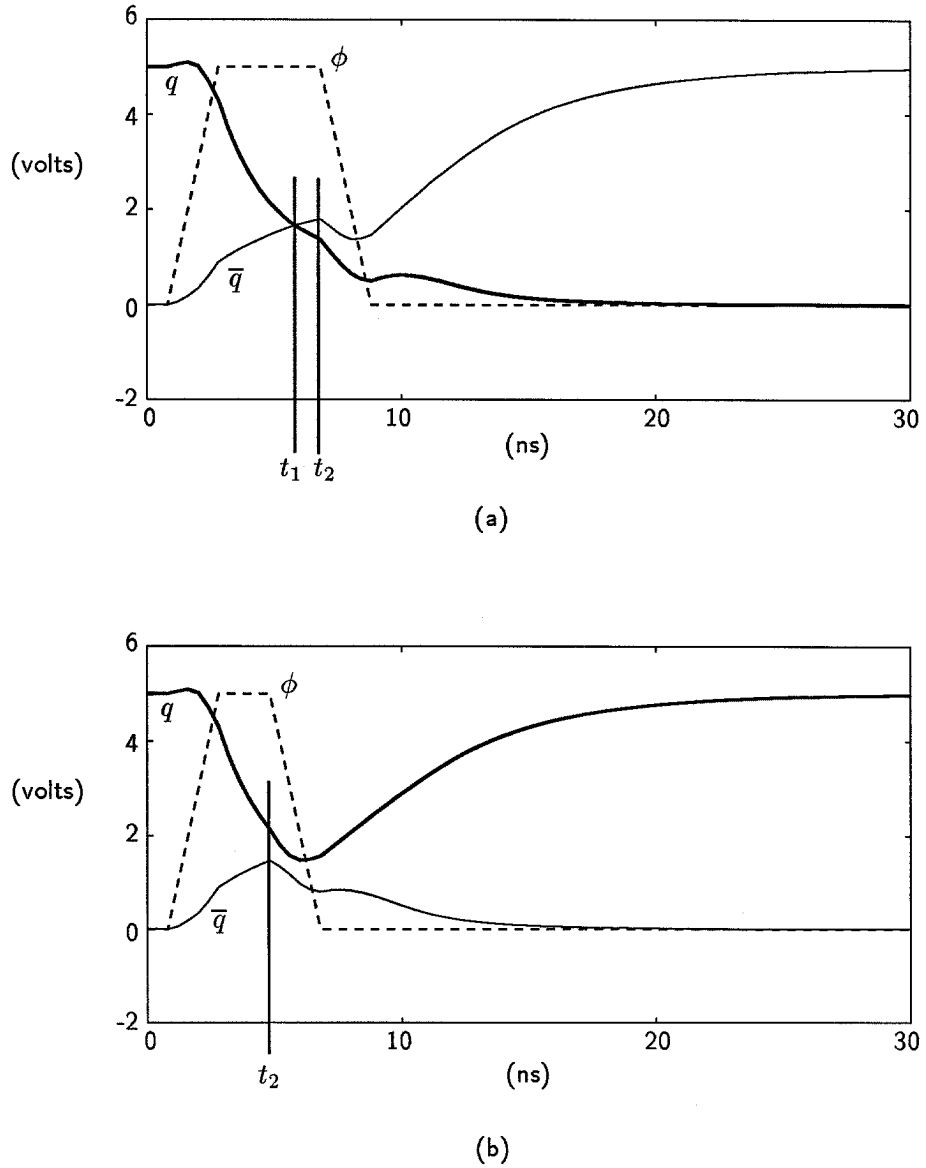

Figure 4.2 Data transfer in CSRL shift-register. Spice simulation of data transfer in shift-register shown in Figure 4.1: (a) successful transfer; (b) unsuccessful transfer.

Multiplexer Functions

There is a class of functions that can be implemented by merely routing one of several sources of data to a given stage. The most familiar example, shown in Figure 4.3, is the exclusive-or (XOR). In this figure and in all other circuit diagrams, I use a rectangle to represent a CSRL flip-flop (one-half of the shift-register stage shown in Figure 4.1). Here, the data are passed unchanged if the control signal is a zero, or data are interchanged with their complements if the control signal is a one. The dynamics of a transfer is identical to those shown in Figure 4.2, except for the lengthening of t_1 because of the higher resistance of the string of pass-transistors, their gate–channel capacitance, and the capacitance associated with their sources and drains. Once these capacitances exceed those associated with the flip-flop, the time t_1 increases as the square of the number of series elements.

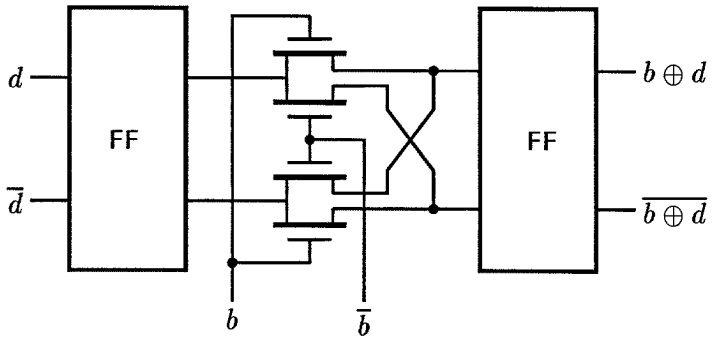

Figure 4.3 Exclusive-or (XOR) circuit. Data are routed from one CSRL flip-flop to another.

Generalized Form

The most interesting property of CSRL is that it can restore signals reliably from very small differences. We can take advantage of this property to simplify greatly the design of sequential logic. A general sequential form is shown in Figure 4.4. The two switching functions, labeled SF_1 and SF_2 in the figure, are general series–parallel networks of n-channel transistors. Gate signals cannot come from within the networks themselves; they are inputs from other CSRL stages clocked on the opposite phase. Each switching function is formally defined as the Boolean condition on input variables under which there is an electrical path (through "on" transistors) connecting its two terminals. In this form, there is no possible path from either input to a logic one. If SF_1 is true, there will be a path from the upper rail to ground. If SF_2 is true, there will be a path from the lower rail to ground. If neither switching function is true, there will not be a path from either rail to ground. If the stage is storing a logic one (upper rail high) and SF_1 becomes true, the upper rail will discharge to ground during the clock high period. By the falling edge of the clock, both rails will be low. The proper operation of the stage depends on a gradual falling

edge on the clock. As the clock begins to fall, current flows through the power-down transistor into both p-channel devices in the flip-flop. Both rails begin to charge toward V_{dd}. The upper rail, however, has a path to ground through SF_1. It loses some of its charge through this path, because the pass-transistors are still on. If the charge lost through this path is greater than any residual difference in voltage left over from the previous state, the lower rail will dominate in the race for charge and will cut off the upper rail's supply. Once again, the sense-amplifier action changes a difference in charging rate into a reliable and unambiguous decision.

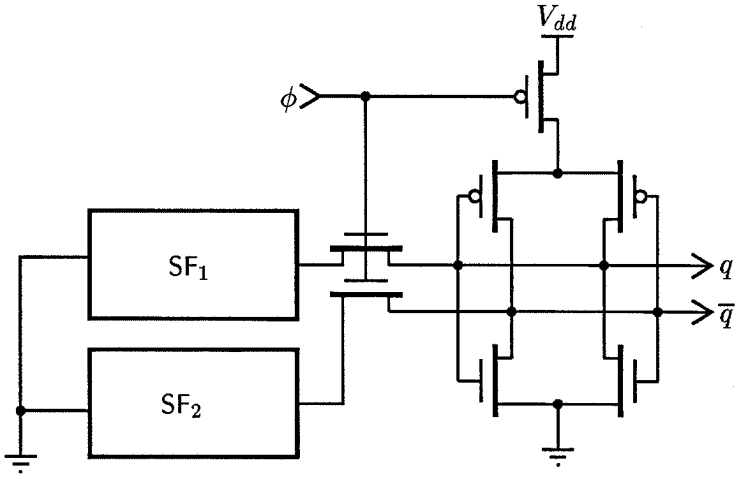

Figure 4.4 General sequential form.

The proper operation of the circuit of Figure 4.4 is shown in Figure 4.5. For this example, both switching functions consisted of four transistors in series. To test the circuit under worst-case conditions, all transistors but those closest to ground were turned on, and the nodes between them were charged to the previous state values. The bottom transistor in SF_1 was then turned on and the clock period was initiated. In Figure 4.5, the residual charge keeps the bottom rail below the top rail throughout the clock rising edge and steady high period. As the clock begins to fall, the path to ground reduces the rate at which the top rail can rise. The bottom rail starts from behind but wins the race. For longer clock-fall times, the crossover occurs at lower voltages, and margins improve markedly.

There are four possible combinations of the two switching functions. The outcome when both are true is not defined. The outcome when both are false, however, is reliably the previous value. The time during which the circuit must store its previous state dynamically on the capacitances of the flip-flop internal nodes is only the clock high time. This value is freshly restored each clock cycle. The circuit is fully static when the clock is low. Charge sharing between the flip-flop nodes and the nodes in the switching function networks is possible, as mentioned. If the clock rise is gradual, the effects of charge stored in the switching function networks can be eliminated. As the clock rises, both pass-transistors turn on before the power-down

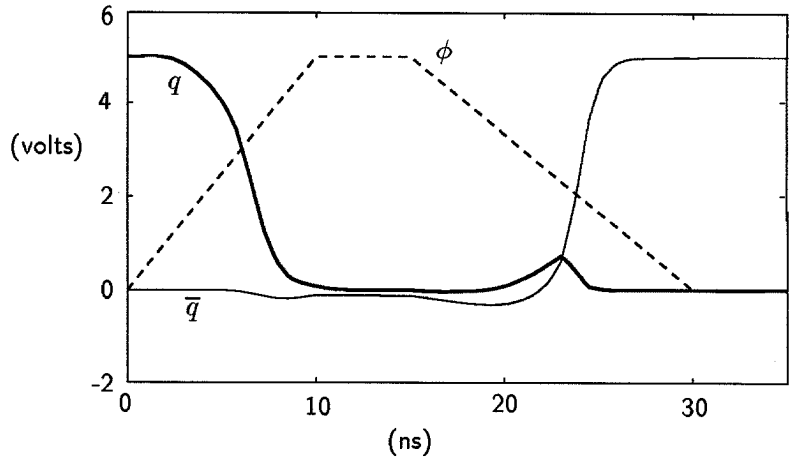

Figure 4.5 Spice simulation of operation of general form. The state of the flip-flop is reversed by connecting the top rail to ground through SF_1 and pulsing ϕ.

transistor turns off. Any stored charge reachable from the flip-flop nodes will be charged to the current state. Because the inputs come from stages clocked on the opposite phase, they are stable during the entire clock event. Hence, any nodes not charged to the current state will not affect the flip-flop nodes, because they cannot be reached. This line of argument constitutes a proof that, given enough time, the flip-flop state cannot be changed unless one of the switching functions is true.

Modification to General Form

Reliable operation of the general, or set–reset, form of the flip-flop relies on the edge of the clock's falling gradually. If the clock falls too quickly, as current flows through the power-down transistor into both p-channel devices, the rail connected to ground through the switching function (SF) will not lose sufficient charge through the SF to allow the other rail to rise to a higher voltage. For reliable operation, there must be sufficient time when *both* the p-channel power-down transistor and the n-channel pass-transistors are on. Another potential problem exists—the node that is *not* connected to ground through the "on" SF is pushed to a value below ground on the falling edge of the clock because of the capacitance coupling between the gate and drain regions of the pass-transistor. This effect is evident in Figure 4.5 as a downward bump in the trace, representing the bottom rail during the falling edge of the clock. Because of this capacitive coupling, the bottom rail must change by a larger voltage before the two rails cross and allow the flip-flop to change state. The capacitive effect ensures that, when the clock falls too quickly, the flip-flop *will not* change state.

The dependence of correct operation in the general form on a gradually falling clock edge is eliminated by a simple modification to the circuit of Figure 4.4. Two p-channel *load* devices are added, each one connected from one input node of the flip-flop to V_{dd}, as shown in Figure 4.6. (In an alternate modification the internal nodes of the flip-flop are connected to V_{dd} through two p-channel transistors.) The gates of the two new transistors are connected to a voltage (V_b) that puts the transistors

near threshold; the transistors supply a small amount of current ($\approx 10^{-6}$ A) to the input nodes because of subthreshold conduction. Figure 4.7 shows the operation of the modified circuit. The node that is connected to ground is not significantly affected by the new source of current—the current is simply shunted to ground, wasting a small amount of power. When the clock is high, the node that was left to float in the unmodified circuit now begins to rise in voltage because of the current supplied by the load. The two rails cross in voltage without a gradually falling clock. The clock may fall instantaneously, restoring power to the circuit and allowing the two rails to snap to the saturated values.

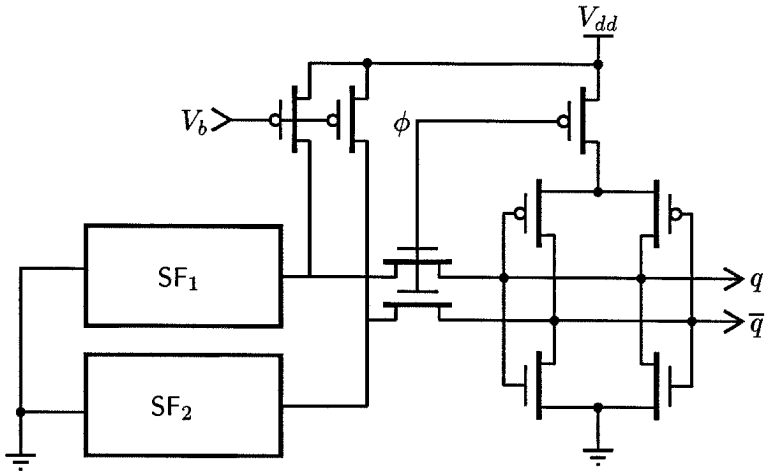

Figure 4.6 Modified general form of flip-flop. Two load transistors are added.

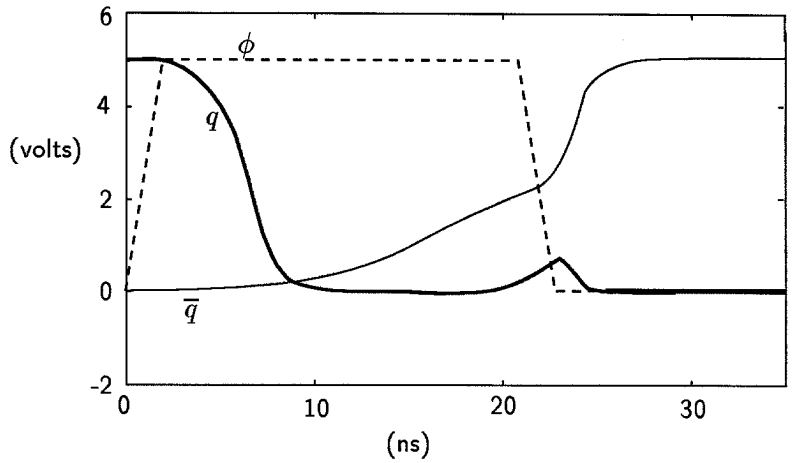

Figure 4.7 Operation of modified general form. The two rails cross without a gradually falling clock.

In addition to eliminating the circuit's dependence on a gradually falling clock edge, the two load devices aid in the static operation of the flip-flop. Previously, the circuit stored state dynamically on the capacitance of the flip-flop's internal nodes when the clock was high and neither SF was on. The load devices perform as pull-ups for the cross-coupled n-channel pull-down networks, forming a stable static configuration.

Semantics

The semantics of a CSRL stage after the clock event can be written in terms of the two switching functions, as follows:

$$\text{NextState} = \begin{cases} \text{SF}_1 \cdot \neg\text{SF}_2 \rightarrow 1 \\ \text{SF}_2 \cdot \neg\text{SF}_1 \rightarrow 0 \\ \neg\text{SF}_1 \cdot \neg\text{SF}_2 \rightarrow \text{LastState} \\ \text{SF}_1 \cdot \text{SF}_2 \rightarrow \text{undefined} \end{cases}$$

.

The semantics assume that the outputs of the stage are used only after the trailing edge of the clock pulse. They do not, however, describe an interesting property of the stage that is exploited in some applications. When the flip-flop is changing state, both nodes are discharged while the clock signal is high. There are thus output transitions on both the rising and falling edges of the clock signal.

The semantics of the CSRL stage has been used as the basis of a simple functional simulator for circuits designed in the CSRL discipline. The simulator was based on the work of Chen on fixed-point semantics and system behavior [CHEN 83]. In most cases, detailed waveforms of circuit operation are not necessary and the functional simulator suffices.

CMOS Implementation of the IPE

Each inner-product element (IPE) consists of 32 stages: $0, 1, \ldots, 31$. There is one simple stage for each bit in the multiplier word B, applied as an input to the IPE. The three clocks, ϕ_1, ϕ_2, and ϕ_3, along with the control signal $load_b$, are distributed to all stages.

A detailed view of one stage is shown in Figure 4.8. Each stage contains an AND function for one bit of multiplication, a flip-flop for one bit of storage for the carry, a flip-flop for one bit of the multiplier word B, and a three-input adder to sum the output of the preceding stage (or the input A in the case of the first stage), with the 1-bit product and the carry from the last 1-bit multiply. At each bit time, the output of each adder, a_{i+1}, contributes to one bit in the final result $A + (M \times B)$.

The detailed operation of the IPE was described in the previous chapter.

The IPE has been implemented using the CSRL design discipline described previously. In the CSRL realization of the stage, every flip-flop is implemented as one-half of the shift-register stage shown in Figure 4.1, and the logic functions are implemented as parallel and series connections of n-channel transistors. This approach results in a fully static circuit with far fewer transistors than are in a more conventional static CMOS design, using D-type flip-flops and logic gates. The use

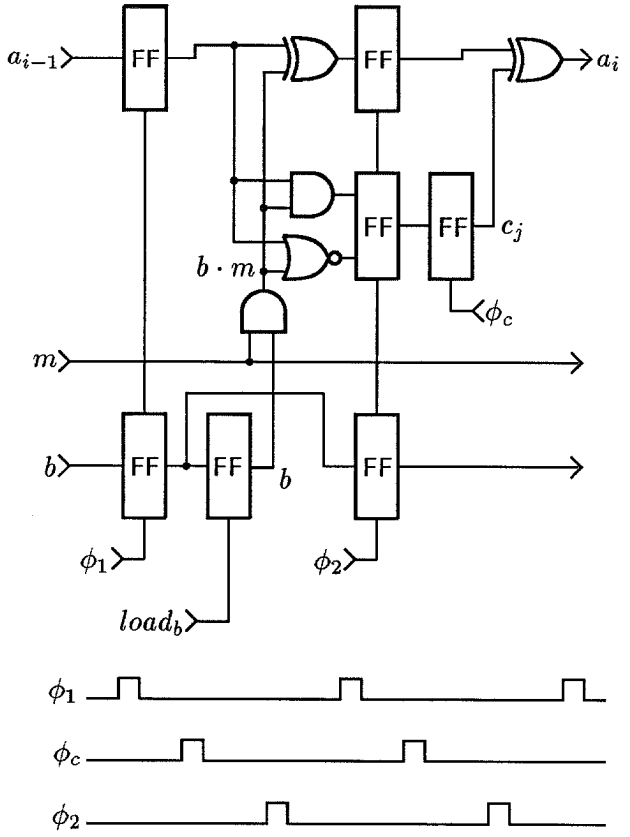

Figure 4.8 Detailed view of logic and timing of IPE stage.

of CSRL flip-flops simplifies the logic design by allowing the use of pass-transistor structures with transistors of all one type.

In each clock cycle of the multiplication and addition operation, each stage of the IPE computes a 1-bit partial product, to be passed on to the next stage, and a carry bit to be used during the next clock period. The well-known truth table for this operation for stage i at time j is shown in Table 4.1.

Table 4.1 Truth table for full-adder.

	a_i	bm_i	c_j	a_{i+1}	c_{j+1}
	0	0	0	0	0
*	0	0	1	1	0
	0	1	0	1	0
	0	1	1	0	1
	1	0	0	1	0
	1	0	1	0	1
†	1	1	0	0	1
	1	1	1	1	1

By inspection of the table we write the algebraic equations:

$$a_{i+1} \leftarrow a_i \oplus bm_i \oplus c_i \quad (1)$$

$$c_{j+1} \leftarrow (a_i \wedge bm_i) \vee (a_i \wedge c_j) \vee (bm_i \wedge c_j), \quad (2)$$

where $\oplus$ is XOR, $\vee$ is OR, and $\wedge$ is AND. We can write the equation for carry by noting the conditions of the input variables under which $c_{j+1} \leftarrow 1$. Alternatively, it is possible to write the conditions where $c_{j+1} \neq c_j$. In particular, we write one expression where the carry bit is *cleared* (reset), $\overline{a_i} \wedge \overline{b \wedge m_i}$; and another where it is *set*, $a \wedge b \wedge m$. These conditions are marked by $*$ and † in the truth table. In all other cases, the carry bit remains unchanged. The expressions for *carry set* and *carry reset* map directly into series-parallel networks of n-channel transistors used in CSRL. For layout reasons, I chose to implement the *carry set* expression as $\overline{a} \wedge (\overline{b} \vee \overline{m})$, and the *carry reset* expression as $a \wedge b \wedge m$, rather than to use gates to generate $b \wedge m$ and its complement. The circuit diagram is shown in Figure 4.9.

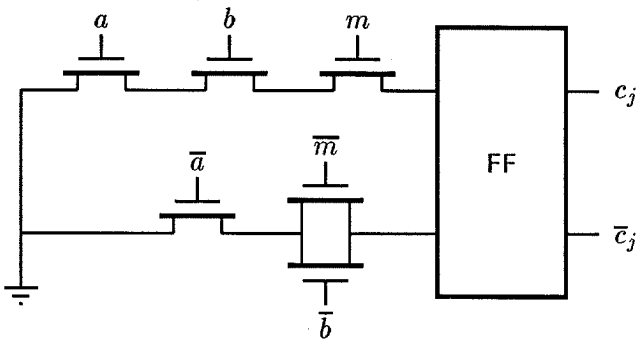

Figure 4.9 IPE carry circuit implemented using the general sequential form.

The sum expression given in Equation 1 is an XOR of three variables and is implemented using two of the four-transistor XOR circuits presented earlier. Figure 4.10 shows the XOR circuit with a minor modification used to generate one-half of Equation 1. As in the carry circuit, the product $b \wedge m$ is generated locally by using a series connection of two n-channel transistors. One transistor is controlled by the signal b and the other is controlled by the signal m. The complement of the product $b \wedge m$ is generated using a conventional CMOS style NAND gate. The NAND output could have been used in the carry circuit to save a transistor, but simplified wiring was a more important concern and led to this combination of choices. One more XOR circuit shown in Figure 4.3 is used to complete the computation of a_{i+1}. In this application, the simple four-transistor XOR circuit is used.

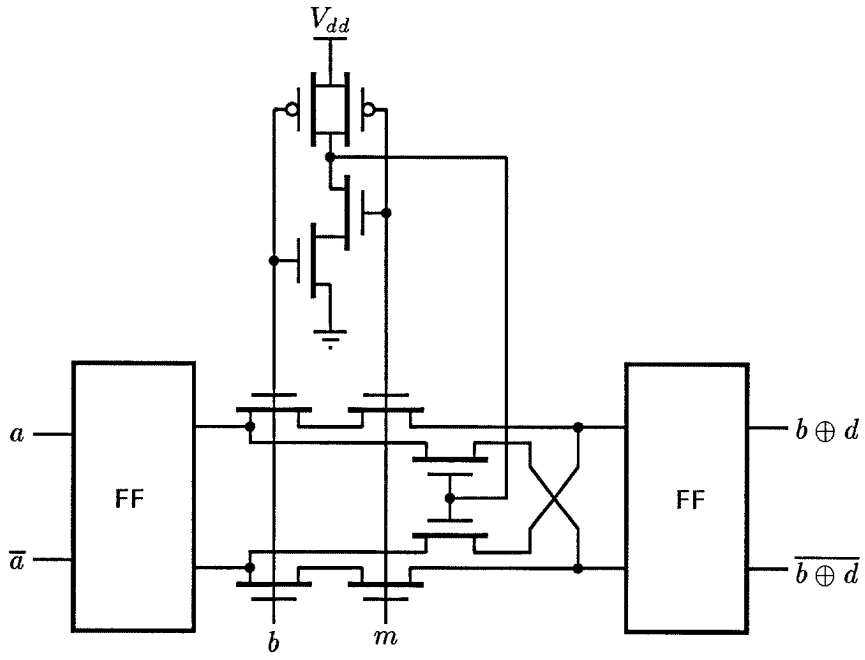

Figure 4.10 IPE sum circuit implemented as a multiplexer circuit.

CMOS Implementation of the Connection Matrix

The connection matrix is primarily a two-dimensional array of flip-flops controlling pass-gates that connect the horizontal and vertical paths through the matrix. The perimeter of the array includes buffers to drive signals through the matrix and logic for loading the state of the flip-flops. Each flip-flop (FF) and pass-gate combination, called a crosspoint, is implemented with a CSRL storage element, as shown in Figure 4.11. The matrix is organized as rows of crosspoints that span the entire array of IPEs and the update matrix. Within each row, one crosspoint is provided for each IPE input and output, and one for each update buffer output, as in Figure 4.12. An entire row of storage elements is loaded at once by driving the appropriate values on the d and $\overline{d}$ lines, then strobing the load line for the entire row. Storing a one in a FF results in a conducting path between the horizontal wire H and vertical wire V.

The row data are shifted in from off-chip serially with a shift-register, driven to all rows of the matrix for loading, then sent off-chip to allow chaining with other chips. The load line (LD) for each row is controlled with a decoder that activates the load signal for one row according to the value on the address lines, on receipt of a load signal from off-chip.

The bottom edge of the connection matrix shown in Figure 4.12 connects to the inputs and output of each IPE and to the outputs of the update buffer. The left edge is used to make bidirectional serial connections off-chip.

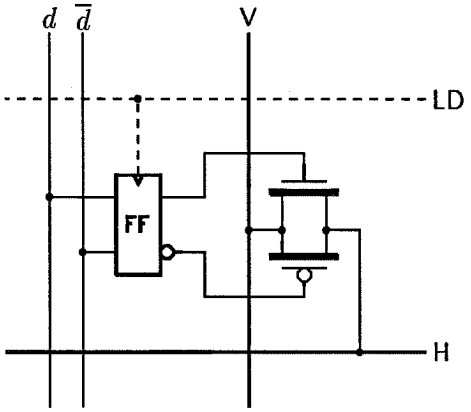

Figure 4.11 Connection matrix crosspoint cell. Storing a one in the FF connects the V wire to the H wire.

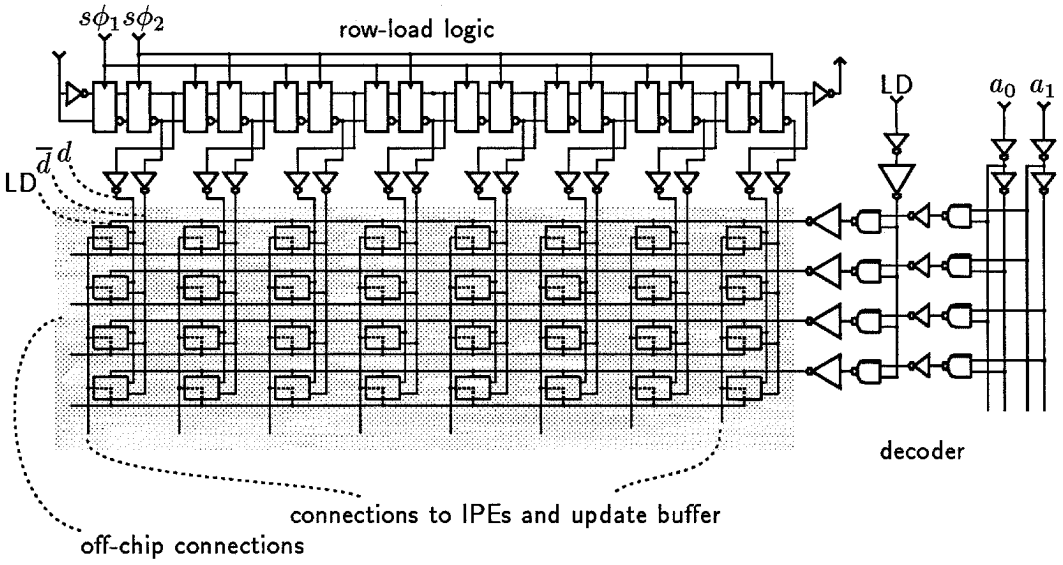

Figure 4.12 Details of connection matrix.

Layout Considerations

The physical layout of the pass-gate at each crosspoint in the connection matrix can have dramatic performance consequences. The transistors making up the pass-gate need to be wide to keep the effective "resistance" of the conduction path low between the vertical and horizontal wires. However, wide transistors channels usually imply high-capacitance source and drain regions. The capacitance of each horizontal wire in the matrix is determined by the sum of the capacitance of the source regions for

all the pass-gates on the wire, even though in normal operation only two pass-gates are active. The same is true for vertical wires. In a large matrix, this capacitance may significantly slow down the transmission of signals. Thus, we must design the pass-transistor layout to ensure high conductively with low source and drain capacitance. The value of drain (or source) capacitance is a function of the area of the drain region (diffusion-to-substrate capacitance) and the perimeter of the nongate region (sidewall capacitance). Sidewall capacitance is a factor only between the active region and the substrate (or well), *not* between the active region and the channel region—making the ring-shaped transistor layout a common choice for low drain capacitance, as in Figure 4.13.

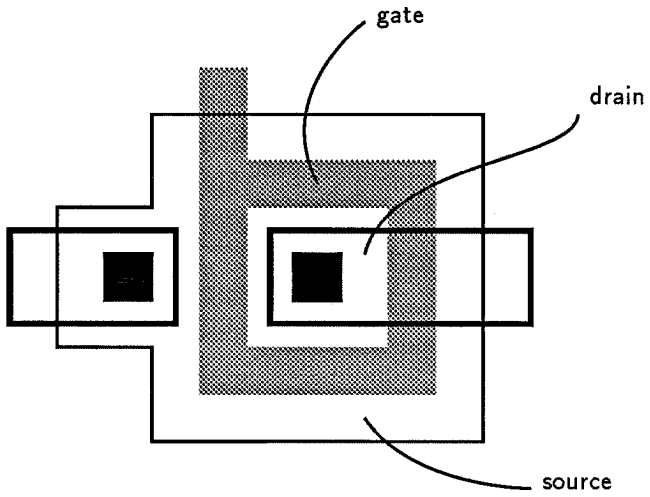

Figure 4.13 Ring transistor layout. This transistor has low drain capacitance but high source capacitance.

In this layout, the width of the channel region is large; however, the drain region has no edge in common with the substrate and therefore has little sidewall capacitance. In addition, the source area is small, minimizing the total capacitance. This transistor, however, has high capacitance associated with its source region. If the source is connected to the power source, the large capacitance provides a stabilizing effect. In our application, *both* the source and the drain must have low capacitance. The double-ring transistor layout shown in Figure 4.14 provides a very low-capacitance crosspoint.
When the transistor is off, it makes a small contribution to the total capacitance of the V or H lines, at the cost of high channel capacitance when the transistor is conducting.

CMOS Implementation of the Update Buffer

The update buffer receives parallel data one word at a time from off-chip, stores

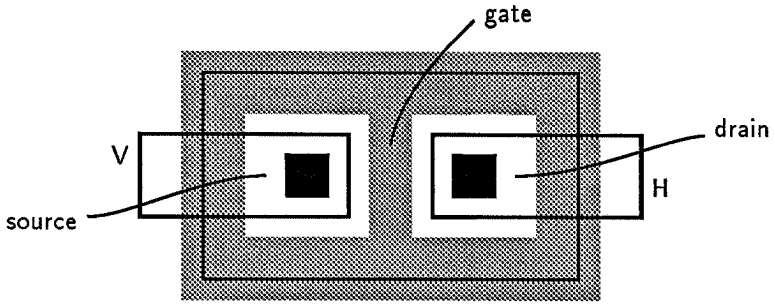

Figure 4.14 Double ring transistor layout. This transistor has low capacitance for both source and drain.

them, then presents them one bit at a time to its interface with the connection matrix. The update buffer, as well as the other blocks of our computing engine, is based on the CSRL FF. Two FFs per bit are employed to achieve double buffering, as shown in Figure 4.15. One FF receives input data from off-chip, while the other supplies data to the IPEs. A global control line transfers data from one FF to the other.

A change in the buffer is made by broadcasting parallel data to all rows. A particular row is selected by a load pulse from the decoder. The load line takes the place of the clock input to the first FF of each stage. A global transfer control (T) takes the place of the clock input to the second FF. It is pulsed from off-chip when all updates have been written. Data are read from all rows simultaneously, one bit at a time, least significant bit (LSB) first. A single "one" shifted through a shift-register and "ANDed" with a read pulse provides the control signal enabling each column to output its values to the bit lines in sequence.

Because the read bit-lines run the entire length of the buffer (32 stages for a 32-bit word) there is an inherently long time constant associated with the wires, and we must take care to ensure a fast read time. This situation is similar to that of RAM design, where a small cell drives a wire the length of the entire cell array. It is impractical to place a large current driver at each cell. Instead, the outputs of the cell are connected to differential lines and an amplifier with high differential gain (sense amplifier) senses the differential value on the bit lines and makes a decision about the state of the storage cell long before the signals are driven to the rails [PRINCE 83].

The differential bit lines and sense amplifier comprise a natural scheme for CSRL style of design. Each CSRL FF produces both data and $\overline{\text{data}}$, and a CSRL FF has high differential gain and thus may be used as a sense amplifier. Figure 4.16 shows the sequence of events for reading one bit of storage. One bit is read by first simultaneously shorting together the two bit lines and connecting them to a fixed reference voltage (V_{ref}), and then pulsing the precharge line (PC). Next, the bit lines are allowed to float until one bit of storage is selected by R. The storage cell pulls one line toward one rail and the other toward the other rail. Then the sense amplifier is pulsed to load it with the value represented by the sign of the difference on the bit lines. Because the bit lines need not reach the rails before the sense amplifier is pulsed, this entire sequence occurs rapidly.

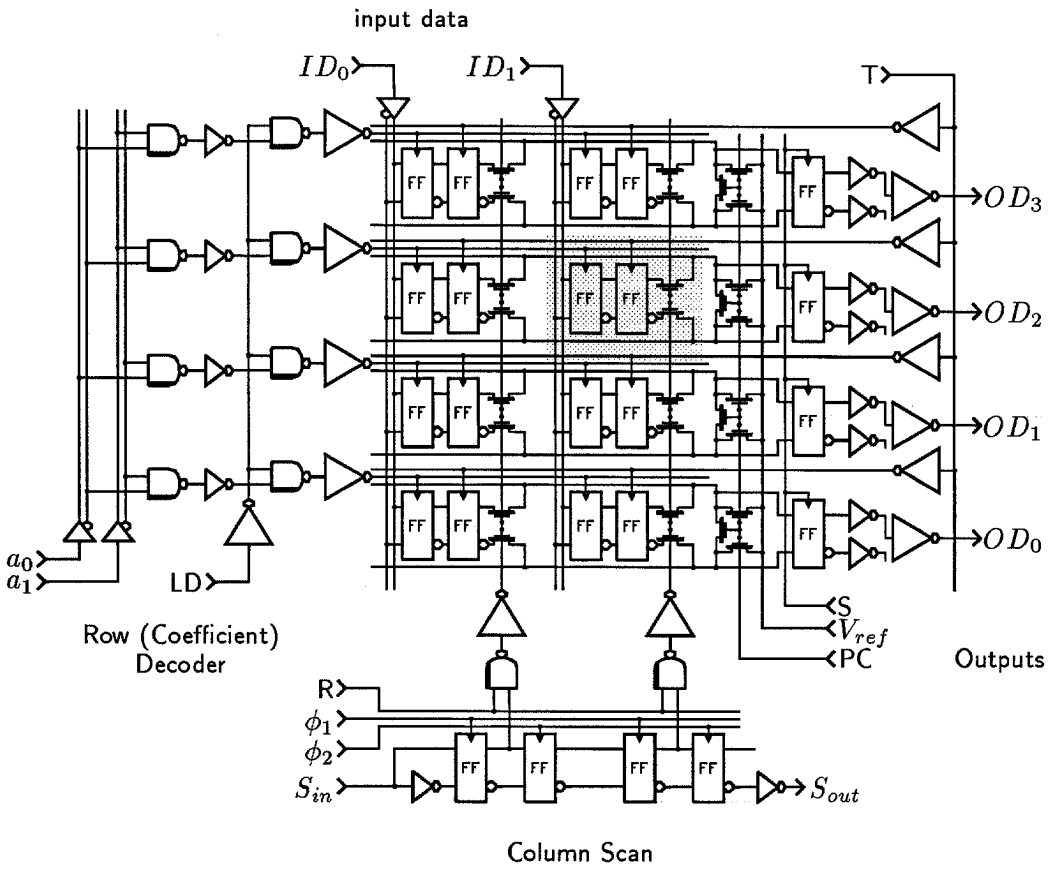

Figure 4.15 Details of update buffer. The version shown has four coefficients of two bits each. The shaded box shows one bit of storage.

In practice, the V_{ref} line may float—shorting the bit lines together ensures that they start at the same voltage, which on the average tends toward V/2. Because the sensing action of the CSRL FF occurs at the falling edge of the clock pulse, no special signal is needed for the clock of the sense amplifier; the same read pulse used to enable the outputs of the FF is adequate. In practice, the precharge signal is driven by ϕ_1 and the read–sense signal is driven by ϕ_2.

CMOS Layout Summary

The cells of the three major blocks of the chip implementing our computing engine were laid out to interconnect by abutment rather than by wires. In some cases, cells were stretched to match up with neighboring cells. Consequently, the size of each block is slightly larger than is absolutely necessary to perform the function of that

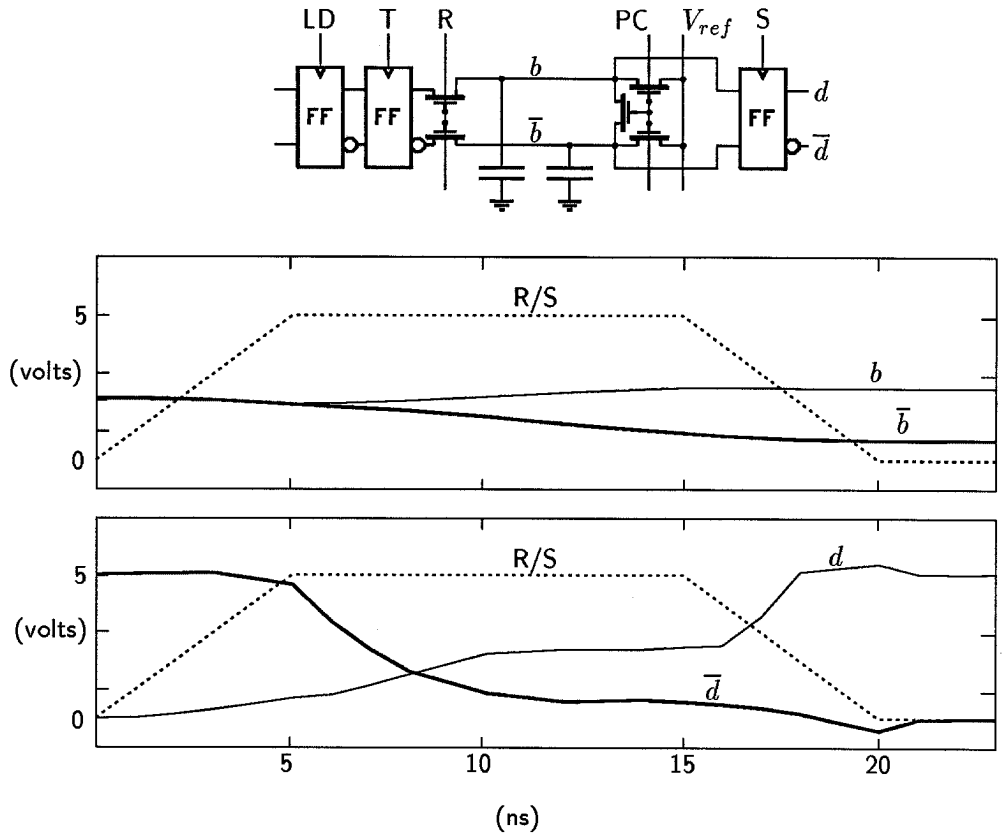

Figure 4.16 Sequence of events for reading the update buffer. Precharged bit lines and a sense amplifier are used to shorten reading time.

block, but the block composition contains no wiring channels and thus is extremely area-efficient. In general, the layout makes efficient use of silicon area.

Figure 4.17 shows the floor plan for a typical version of the chip. Most of the area is taken up by the processing elements and the connection matrix. The sizes, in lambda [MEAD 80], shown in the figure are for a 32 IPE array of 32 bits each, with 32 words of update buffer capacity, and a 44-channel connection matrix. The largest version of the chip fabricated to date was an experimental version with this configuration, but with a connection matrix of 64 channels.

Figure 4.18 shows a photograph of the fabricated chip.

NMOS Implementation

The processing elements called UPEs in the previous chapter were implemented in *n*MOS technology. The CMOS design for the IPE was a simplification and refinement of the *n*MOS UPE design. The UPE implementation was useful for developing the basic musical-instrument models, and the chips are still in use in a music demonstration system.

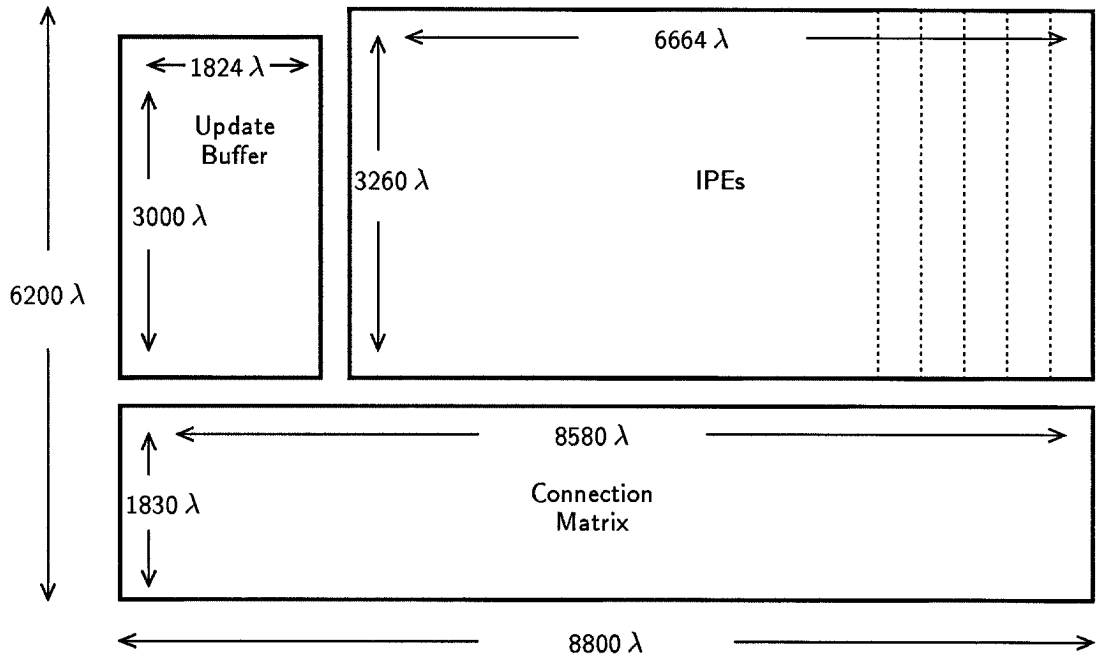

Figure 4.17 Layout dimensions of chip. Dimensions are not shown for pads.

The nMOS UPE differs from the CMOS IPE in several important aspects: (1) the nMOS UPE uses dynamic storage of state (dynamic FFs), whereas the CMOS IPE is fully static; (2) the nMOS UPE includes pipeline delays between stages, whereas the CMOS IPE does not—it uses global lines running the length of the array; (3) the nMOS UPE performs the function of the CMOS IPE and also includes a mechanism for performing linear interpolation; and (4) the nMOS UPE performs 64-bit adds.

The UPE nMOS circuits followed common nMOS design practice as presented in Mead and Conway [MEAD 80], and therefore are not described here.

Each UPE stage measures 70 λ by 186 λ. The 32-bit array measures 2310 λ by 186 λ, with sign-extension circuitry accounting for the extra space.

Conclusion

I have introduced a new logic form and clocking scheme for CMOS integrated circuits, called CSRL. Systems implemented with CSRL feature fully static operation and simpler circuits than do those using conventional CMOS forms. I have used the new logic form to implement our architecture for sound-synthesis. The result is a fully functional chip, integrated into a system for experimentation with sound

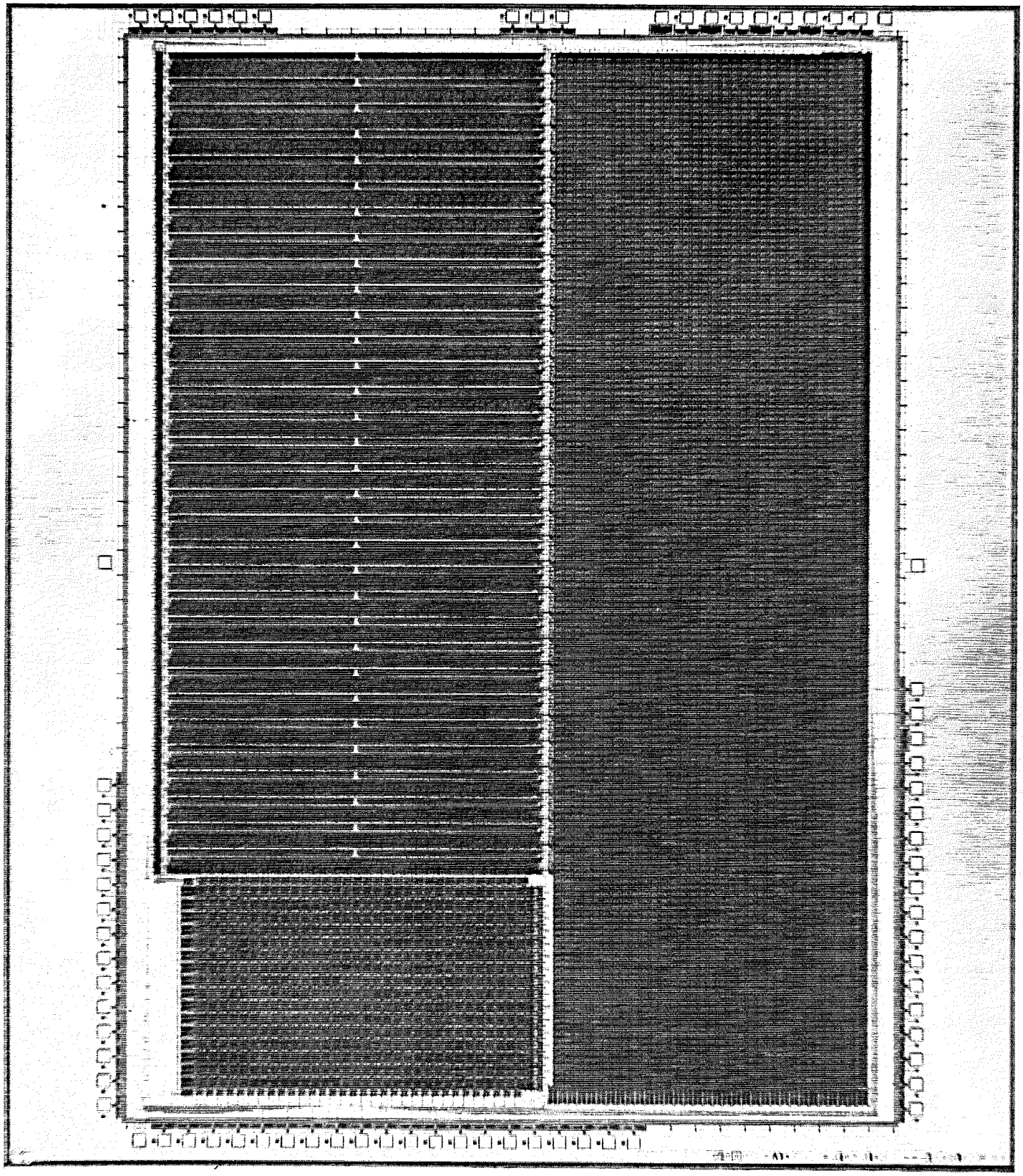

Figure 4.18. Photograph of fabricated chip.

synthesis algorithms. An nMOS version of the array of processing elements has also been implemented and is being used to generate musical sounds.

Chapter 5

Other Architectures

In this chapter we examine alternative architectures for musical sound synthesis. Two important aspects of the design are investigated: serial versus parallel arithmetic and the interconnection scheme. Three architectures are presented and compared, the results are summarized in the concluding section.

Our computational task is clearly dominated by the multiply computation and it is therefore tempting to focus a study on the multiply algorithm. Many structures have been invented for performing computer multiplication, spanning the entire range of tradeoffs between area and speed. The complexity of the multiply computation has been a popular subject in the literature. One measure commonly applied to multiplication is the space–time squared product. The theoretical lower bound for multiplication of n-bit integers, has a space–time squared product of $AT^2 = O(n^2)$ [ABELSON 80], [BRENT 80]. A *simple* realization of a n-bit multiplier with this complexity is not yet known [LUK 81]. The bit-serial multiplier proposed by Lyon and others [LYON 76], [JACKSON 68] has a space–time squared complexity of $AT^2 = O(n^3)$. Most researchers have concentrated on minimizing the *time* (latency) required for multiplication. In our application, and in many others, *throughput* rather than *latency* is the important issue. Pipelined multiplication structures are efficient in these applications because many independent operations need be computed and they may be computed simultaneously. In these systems one attempts to minimize the cost/performance, AT/p, where p measures the number of problem instances within the structure [SEITZ 84].

Because of our freedom to trade time and space in an attempt to maximize total system throughput many alternatives exist in the choice of multiplication structures. This chapter presents architectures based on three varieties of multipliers.

Arithmetic is not the entire story, however, and here is where a complexity analysis breaks down. The multiplication algorithm has to be studied in the context of the problem at hand and in the context of the entire machine architecture. In our application, equally important to the multiplication algorithm, are the means of storing intermediate results and coefficients, and the communication of results within the system. While space–time products are important theoretical measures to the

limitation of computing machines, practical machines must be analyzed *in toto* and with respect to specific problems. This analysis is difficult to do in general however, because only poor models exist for area and energy of computations and communications, but there has been some success [THOMPSON 80]. Often the issues of practical importance are technology-specific. Therefore, in this chapter, rather than attempting a theoretical analysis of various architectures, we will simply present one architecture representative of each different class of solutions and discuss the issues associated with the design of each, and the relative strengths and weaknesses of each approach.

In this section we redefine our target task, which will be used as a guideline in machine architecture development. The task may be represented as a graph of computation nodes (UPEs), where each node accepts three inputs, A, B, and M and produces the result $Y = A + B \cdot M$, along with an optional delay of one sample time, notated $Y = (A + B \cdot M) \cdot Z^{-1}$. In the case that a node generates a delay, it contains state. In practice, it is necessary to produce other functions at each node, for instance, mod or limit, but here we will ignore them. One important simplification of the problem is that the topology of the graph does not change within the course of the computation. This is evident from the fact that there are no "if than elses" or data-dependent operations in the task. The graph is "solved" once each sample time by computing the output of each node, given the inputs, and making each output value available for input to other nodes. In the case where a delay is specified, the output value is stored and made available on the next sample time. Inputs that do not come from other UPEs are supplied externally and are called *coefficients.* Coefficients change approximately 1000 times more slowly than the sampling rate, for they usually reflect user-driven changes such as the striking of a key on a keyboard or modifying the blowing strength in a wind model. In our applications we have found that on the average in a graph there exists about one coefficient per UPE. That fact determines some design decisions. Furthermore, music applications require that all changing coefficients in a graph change simultaneously. Of course, some coefficients are static during the course of the computation.

Perhaps the most important aspect of our task is that it must be solved in real time—the graph must be solved in its entirety each sample time. This constrains the possible set of solution strategies. We would like to design a machine that can solve a graph in real time. Whereas it is perfectly acceptable to limit the size of the graph for any of the machine, the machine architecture must be expandable to handle arbitrarily large problems by adding more hardware. This constraint forces us to look at strategies using concurrency, where more that one processor are working simultaneously on various subgraphs. The limit of computing separate pieces of the graph simultaneously is to assign one processor per graph node. In this implementation each processor needs only to complete one computation per sample time, whereas in a coarser division of the graph, one processor is responsible for a number of graph nodes and therefore must be able to complete the computation of each node in a sufficiently short time to be able to complete all of its nodes. In the coarser division of the graph, memory is required to store the outputs of UPEs with state, in order to delay using the new results until the next sample time. Both these strategies are possible with today's VLSI technology, and each offers distinct ways

to exploit various structures for computation and communication of results within the graphs.

Designing a machine for our task in general is an important one, for it represents the classic tradeoff in VLSI design. We can build many slow, small processors working in parallel or one large fast processor "emulating" the work of the parallel collection. The architecture we implemented has previously received little research attention and takes the slow small processor approach. Both approaches have their merits, however, and in this chapter we will investigate these two approaches, along with the next logical step in trading off speed for area.

A Serial–Serial Architecture

A serial–serial approach to multiplication uses minimal hardware at the expense of slow computations. Using a minimal hardware configuration, the serial–serial approach requires approximately n^2 steps to complete. In sound synthesis applications, where the sample rate is 50 KHz, the bit clock rate required to complete one multiplication in one sample period is $32 \times 32 \times 50$ KHz, or approximately 50 MHz. Such a clock period is higher than current commercial designs but is within the limits of present day technology. Because the clock rate needed to perform 32-bit multiplications at audio sample rate is near the maximum of the technology and the approach requires minimal hardware, serial–serial multiplication is well matched to the sound-synthesis task.

Figure 5.1 shows a structure for serial–serial multiplication and addition of positive integers. At the core of the structure is a full-adder and a carry flip-flop. The two numbers to be multiplied are placed in the m and b registers, and the number to be added is placed in register a.

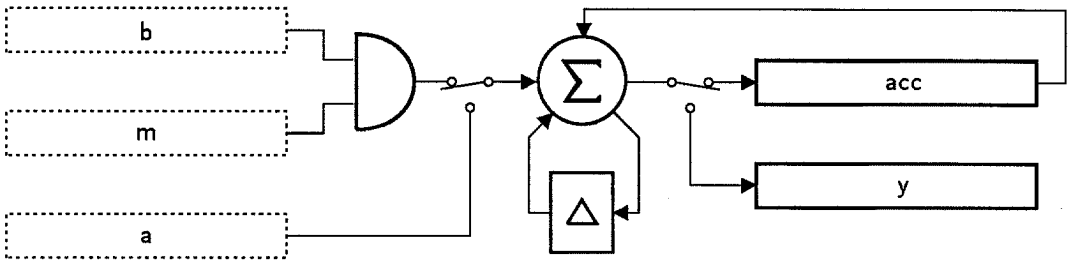

Figure 5.1 Serial–serial multiplier–adder.

The control multiplication and addition follows:

```
for i from LSB to MSB
    clear_carry;
    for j from LSB to MSB
        ⟨acc[j-1], carry⟩ ← FA(b[i] AND m[j], acc[j], carry);
    end j
end i
clear_carry;
for j from LSB to MSB
    ⟨y[j], carry⟩ ← FA(a[j], acc[j], carry);
end j
```

The inner loop generates one n-bit partial product for each bit of b as it adds in the sum of the previously generated products, held in the accumulator (acc). In multiplication algorithms, each successive partial product carries two times the weight of the previous one. Therefore, the newly generated bits are placed into the acc register with an offset. After all n bits of b have been used, the acc register holds the high n bits of the result. The low bits of the result are lost, but could be retained, with the use of an additional register. After n^2 cycles, the switches in Figure 5.1 are moved to the lower position, and the structure is clocked an additional n cycles to add the a to acc and place the result in y.

The structure can be modified to work with two's complement numbers by noting that the high bit of b has negative weight. The acc must be subtracted from the new partial product on the last bit of b. In practice, the input to the full-adder from acc and the output to acc are inverted when $i = n$.

The serial–serial multiplier structure may be used as the processing element (PE) in a VLSI system for sound synthesis. As with our architecture based on serial–parallel multipliers described in Chapter 3, *A VLSI Architecture*, PEs are placed in a linear array and connected via a connection matrix or *switch*, as shown in Figure 5.2. Each PE must store locally the acc and y registers, shown as solid boxes in Figure 5.1. The input registers, shown as dashed boxes, do not exist as part of each PE site. These inputs are provided from one of three sources: (1) from the y register of other PE sites, (2) as coefficients provided from the external host, or (3) from the output of other chips. The specific origin of each of the three inputs is programmed through the switch.

On average, in our applications, we have found that one coefficient per PE is needed. The application also requires that the coefficients be double-buffered to prevent instability by changing coefficient values in the middle of a computation. In this design, a double-buffered coefficient is provided at each PE site, where it may be wired to provide input to any of the PE inputs, or it may be exported to another site. Interleaving the coefficient registers with the PE registers places additional constraints on the assignment of computational nodes to PEs, but we assume that, in general, coefficient usage is evenly distributed throughout the computation graph, and a few extra local channels in the connection matrix will be sufficient to satisfactorily distribute the coefficients. A pessimistic approach would provide one additional wiring channel for each coefficient, running the length of the matrix.

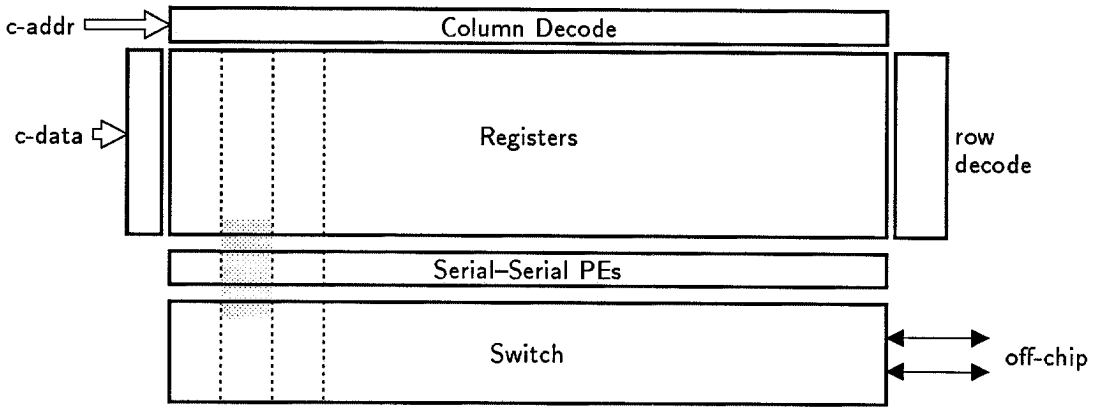

Figure 5.2 Serial–serial architecture. Serial–serial processing elements are placed in a linear array and connected through a connection matrix, or switch.

The most straightforward way to implement the y, acc, and coefficient register (c register) at each PE site is as shift registers. Random access memory (RAM), however, is more efficient because only one bit of storage is needed in a RAM approach versus two bits of storage per bit for shift registers. A shift register can be simulated using a RAM by successively reading and writing bits of the memory. A read-only version of this idea was presented in conjunction with the update buffer presented in Chapter 3, *A VLSI Architecture*. Figure 5.3 shows one bit of memory structure for one PE site. This illustration represents one particular bit, say the LSB, for the three registers at each PE site. The structure is replicated in the vertical direction for other bits of the word and in the horizontal direction for other PEs.

On every cycle, one bit is read from each of the c, acc, and y registers and one bit is written to acc and y by first reading one entire row from the memory array, then writing another. Because all the PEs are working in synchrony, the memory control logic is shared by the entire array. The memory control is as follows:

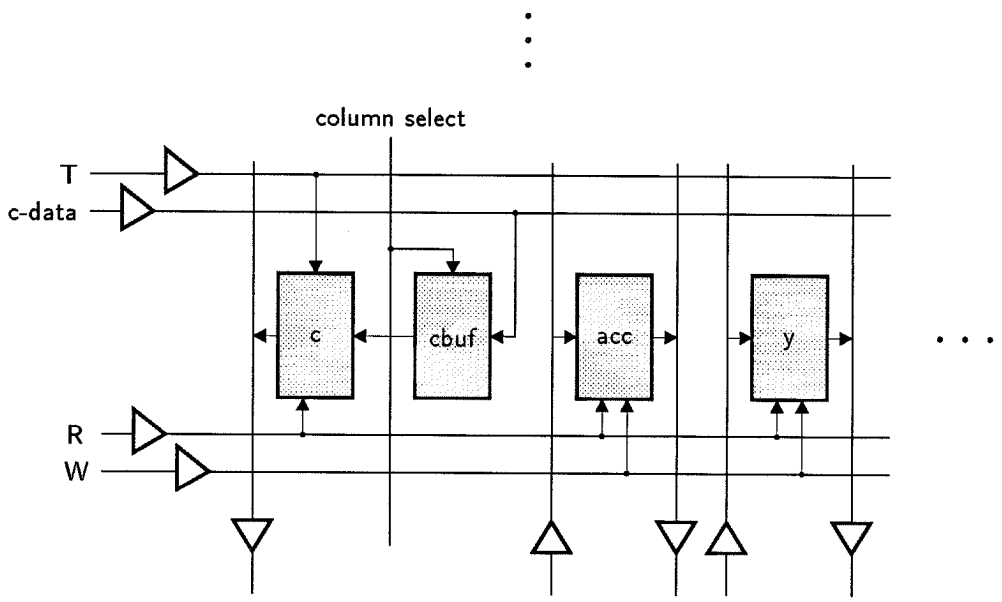

Figure 5.3 Memory structure for one bit of one PE site. RAM is used to implement shift registers. In addition to an accumulator register and an output register, each PE site has one coefficient register.

```
for i from LSB to MSB
    "read next b"
    read_row[i];
    for j from LSB to MSB
        read_row[j], compute next bit, write_row[j-1];
    end j
    "sign extend acc"
    write_row[MSB];
end i
"done with multiply"
"add"
for j from LSB to MSB
    read_row[j], write_row[j];
end j
```

The c register is loaded in parallel asynchronously with the operation of the PE. Address lines are used to load data from the c-data lines to each cbuf flip-flop for all bits of the word. The data are transmitted to the associated c flip-flop with the T signal. Figure 5.4 shows the details of the connections between the registers, the PE, and the switch. The data in the c register are read bit serially and are used as input to the switch. The acc register in each slice is connected directly to the PE.

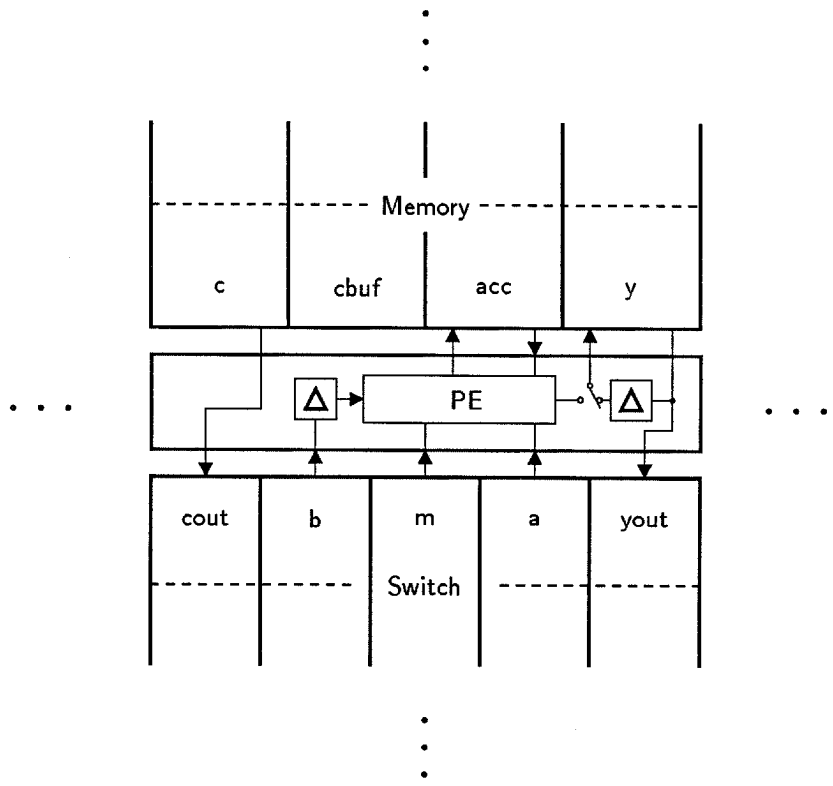

Figure 5.4 Details of the connections at each PE site.

The y register's output goes to the switch for broadcast to other PEs but its input comes from the PE or is recirculated.

The switch provides bit-serial communication among PEs and bidirectionally with the outside world. It is functionally identical to the one presented in Chapter 4, *A VLSI Architecture*, with differences in the number and nature of channels. One channel with local wires, spanning the width of one PE site, is provided to accommodate the connection of the c register at a PE site to a PE input at that same site. Also, fewer long distance channels are needed because the coefficient buffers are interleaved with the PEs.

VLSI Implementation

In this section I discuss issues involved with the VLSI implementation of the serial–serial architecture. This architecture has not been implemented, as has the serial–parallel one; therefore, I will *estimate* its speed, area, and power requirements. Because CMOS is a widely available technology for VLSI systems, I will use it to illustrate the implementation. Within CMOS technology, many options exist for circuit design; one such circuit design methodology is presented in Chapter 4, *VLSI Implementation.* Any particular circuit design style has its own implications as to circuit size, speed, and power dissipation. In this study, as in the implementation of the serial–parallel architecture, I have chosen standard CMOS technology and *static* circuits. It is always possible to use special processes to help produce more

efficient chips; for instance, a process with high-resistance polysilicon reduces the area consumed by static RAM, and a floating-gate technology is useful in reducing the size of the connection matrix. However, special processing techniques do not give any architecture a clear advantage over the others, so we will use a standard technology for this comparison.

Pipelining. Based on a sample rate of 50 KHz, the target clock rate is $50K \times (32^2 + 64) = 54.4$ MHz, corresponding to a period of 18.38 ns. Referring to the memory control algorithm the system must read memory, propagate the data through the switch, through the PE and finally write data back to the memory. According to simulation results, with current technology, 18 ns is not sufficient time to complete this sequence of steps. The sequence of steps can be pipelined to achieve concurrent operation of various pieces of the system and relieve some of the speed burden. The memory read of a bit and its propagation through the switch can proceed simultaneously with the computation involving the previous bit and its subsequent writing to memory. Pipelining in this manner has three consequences to the implementation: (1) the memory control algorithm must be modified to synchronize the reading and writing of data—including adding a step to fill the pipeline initially, and consequently a slightly shorter period, (2) the memory must have one part for reading and a *separate* one for writing, and (3) an additional storage element must be added at the receiving end of the switch to isolate the two halves of the pipeline. Only the change to the memory structure has any significant impact on the implementation; the other two are minor. Dual-ported memory in general increases system speed at the cost of a larger memory structure. In this case the tradeoff is necessary because the system does not perform to speed without it. The memory shown in Figure 5.3 is dual-ported.

Dual-rail. One concern with designs employing a connection matrix, or switch, is the propagation delay through the switch. One way to shorten the delay through the switch is to precharge the data lines and use a sense-amplifier at the receiving end to restore the data quickly to logic levels. Although single-rail sense-amplifier schemes are known, a dual-rail scheme fits well with our choice of static memory cells. Dual-rail data lines emanate from the memory and it is possible to extend the lines through the matrix and place a sense-amplifier at the receiving end of the switch. This scheme increases the number of wires running through the cross-point cell in the matrix by one in each direction but does not increase the number of pass-transistors, because only one transistor per line is required for differential signals versus two in a single-rail scheme.

Layout Area. In this section I estimate the area of one slice through the floor plan shown in Figure 5.2. One slice represents the area consumed by one processing node and is therefore a basic measure for the total system area. The layout area of the processing node can be used to plan the layout for an entire chip. The area of the cells in the node slice is based on prototype designs and is not exact but is within approximately 15% of what is expected in a final layout. All dimensions are given in lambda units [MEAD 80].

The width of the slice is dominated by the width of the cross-point cells for the switch. The cross-point cell contains a flip-flop, two pass-transistors, data paths for loading the flip-flop, and conductors for transmitting data through the switch. The cross-point cell is 40 by 50. Therefore the width of one slice is set at $4 \cdot 40 = 160$. We will include 10 channels in the connection matrix; therefore, the switch contributes 500 to the height of one slice. The height of the register is 40 per bit for a total memory height of 1280. Column decode and PE are about 100 and 40 high, respectively. Combining all these dimensions, the area estimate for an entire PE slice is 160×1920.

A linear array is formed by lining up the slices in a row. Consider a process where $\lambda = 0.8\mu$. In such a process, a 10×10 mm die contains approximately a usable area for PE slices of $11000\,\lambda$ square, reserving the remaining area around the perimeter for pads and wiring. The usable width allows for an array of at least 64 PE slices. But one row uses less than 20% of the available vertical dimension. Multiple rows must be laid out to utilize the available area; in this case, six rows are possible for a total of 384 PEs. Separating the processing elements into rows creates the need to route wires between switch sections of adjacent rows. This problem is similar to the one faced when separating PEs at chip boundaries. Because the number of channels in the switch grows as the log of number of PEs, a relatively small number of wires suffice and are ignored in further calculations.

Power Consumption. In this section I estimate the power requirements for this implementation. Ultimately, I will compare the requirements with those of other architectures. For the purposes of the comparison only the power used for computing and transferring data *within* the chip is considered. Communication between chips, loading of coefficients, and initialization of the connection matrix are constant among the different architectures and are ignored.

Power dissipation in CMOS circuits arises from two mechanisms [GLASSER 85]. The first is due to the charging and discharging of nodes. When a node is charged, energy is dissipated in the transistors connecting it to V_{dd}. The total energy dissipated is independent of any details of the pull-up transistor (or transistors) and the path the voltage takes. An equal amount of energy is dissipated when the node is discharged. This energy having to do with charging and discharging nodes is called the *switching energy*, E_{sw}. The energy stored on a capacitor is qv; therefore, the total energy required to charge and discharge a node is

$$E_{sw} = CV_{dd}^2 .$$

If we assume the the node is switching at a frequency of f, the average power consumption is

$$P_{avg} = fCV_{dd}^2 .$$

The other source of power dissipation in CMOS circuits is due to transistor conduction overlap. Whenever a pull-up transistor (or transistors) on a node is (are) simultaneously conducting with a pull-down network, a current flows directly from V_{dd} to ground. The excess power due to conduction overlap can be large in some cases, for instance, when reading a value with a small inverter from a large bus.

Experiments have shown, however, that under normal circumstances the overlap current is relatively small compared with the current that goes into charging and discharging nodes; therefore, for the purposes of the comparison, we will ignore it.

The two primary sources of the capacitance of the nodes of a circuit are (1) the capacitance between interconnection wires and substrate, or other conductors, and (2) the capacitance associated with transistor gates and source–drain area and perimeter. Based on recent MOSIS measurements [MOSIS USER'S MANUAL 86], and $\lambda = 0.8\,\mu$, we assume the following values:

$$\begin{aligned} \text{interconnect} \quad & C = 6.0 \times 10^{-4}\,\text{pF}/\mu^2 = 4.0 \times 10^{-4}\,\text{pF}/\lambda^2 \\ \text{transistor} \quad & C = 0.3 \times 10^{-4}\,\text{pF}/\mu^2 = 0.2 \times 10^{-4}\,\text{pF}/\lambda^2. \end{aligned}$$

Because all the power goes into switching nodes, computing the power consumption is simply a matter of summing the products of every node size with its switching frequency. In the serial–serial design we consider only the power associated with the PE slices, ignoring decoders.

The static RAM structure is written by selecting a row and driving the column write bit-lines to overpower the selected cell. The predominant energy goes into charging the bit lines, because the switching energy associated with the cell is relatively small. In this design every bit line is driven on every cycle. However, on average only one-half of the lines change state each cycle, assuming an even distribution of ones and zeros. These nodes are either charged, discharged, or neither, but not both on each cycle; therefore, their power consumption is $\frac{1}{2}fCV_{dd}^2$. There are two lines per column in the RAM, so we count one line for each column. Reading is done by precharging the read bit lines, allowing the cell to drive the lines, then sensing the difference. The precharged lines end up at restored logic levels at the end of each cycle. Prior to the precharge, one-half of the lines are already at the precharge level, so we count each pair of lines as one during the precharge phase of the cycle. During the discharge phase, only one of every pair of lines is discharged, so again count each pair as one.

The RAM read bit lines are precharged through the connection matrix to the receiving PE slice. We assume that every line in the connection matrix is used every cycle, but again only one-half change state at each cycle. The energy associated with the PE itself is small in this case relative to the memory and switch data lines and is ignored.

From the layout information we estimate the total capacitance of the nodes under consideration. Following is a summary of the capacitance calculations:

$$\begin{aligned} \text{write bit lines} &: 0.359\,\text{pF} \\ \text{read bit lines} &: 1.08 \\ \text{vertical switch data lines} &: 0.600 \\ \text{horizontal switch data lines} &: 0.448 \\ \text{total capacitance} &: 2.49\,\text{pF}. \end{aligned}$$

Therefore, the energy per slice per cycle is

$$E_{slice} = 62.3 \times 10^{-12}\,\text{Joules}.$$

at $V_{dd} = 5.0$ volts, and the total chip energy per cycle is

$$E_{chip} = 23.8 \times 10^{-9} \text{ Joules.}$$

The energy associated with one slice is expended $n^2 + 2n$ times per operation; therefore, at $n = 32$, the energy per operation is

$$E_{OP} = (n^2 + 2n) \cdot E_{slice} = 67.78 \text{ nJ.}$$

At the designed clock rate of 54.4 MHz, the total chip power consumption is

$$P = 1.30 \text{ Watts.}$$

A Parallel–Parallel Architecture

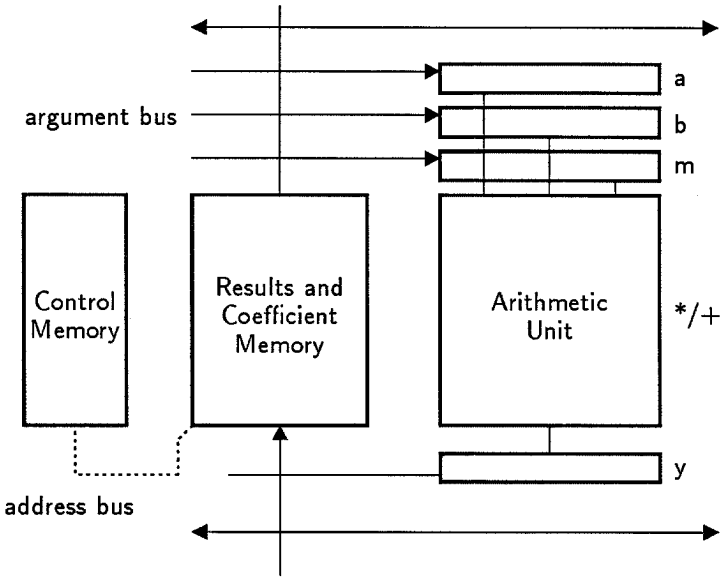

Figure 5.5 Architecture based on parallel–parallel arithmetic. Outputs and coefficients are stored in RAM; control memory supplies a stream of addresses.

The architecture presented in the previous section employs a serial–serial arithmetic algorithm as a way to match the speed of the calculation with that of the application. Another way to match the speed of the calculation to the application is to employ a single fast processor and "emulate" the work of a parallel collection of processors. The basic structure of an architecture based on parallel–parallel arithmetic is shown in Figure 5.5. PE outputs are stored in the right-hand memory,

along with coefficients. On each cycle, three words are read from the memory, the a, b, and m inputs to the multiply–add unit. After the multiply–add operation, the result is stored back in the memory. The *address* of the operands and results for each cycle are stored in a second memory, the *control memory.* This memory cycles through its contents once per sample-time. The control memory also contains extra bits to control routing of data to and from other chips.

The control memory is analogous to the *program memory* in a conventional computer. It also plays a role similar to the connection matrices of the previously presented architectures. A connection matrix arranges the interconnection of PEs in space, whereas this control memory arranges the interconnection of PEs in time. PEs communicate implicitly by one PE reading from a location in memory where another left its result. The correct sequence of reads, writes, and interleaved input–output operations are precompiled offline. As with the connection matrix, the control memory is loaded prior to execution. It is important to note that every chip in the system must have its own control configuration; therefore, the control address cannot be broadcast and shared throughout the system.

The architecture shown in Figure 5.5 suffers from a problem that it has in common with conventional computers with respect to our application. Access to arguments and coefficients is sequential in time. Substantial savings in the cycle-time can be achieved by simultaneously accessing the operands a, b, and m, and if possible by simultaneously storing the previous result y. Several options are available for providing simultaneous access to the memory. The most straightforward scheme for providing simultaneous access to memory is to build the memory with multiple ports. Our application requires three read ports and one write port. Of course, any lesser degree of speedup can be achieved by using less ports to reduce the cycle-time by at most three memory operations. Multiporting is implemented by adding multiple bit lines and select lines to the memory cells composing the memory bank. Also, independent decoders need to be supplied for each port. Because memory cells are dominated by the bit lines and the select lines, adding more ports drastically increases the size of the memory. In our application, a more efficient approach may be to provide concurrent memory access by using independent multiple memory banks.

One way to apply the notion of independent memory banks is to build a separate memory for coefficients, so that coefficients can be accessed concurrently with the other operands to the multiplier. Since most PEs take at least one input as a coefficient, most cycles can be reduced by one memory read. This idea can be extended to the other operands.

Unit Delay Model. For both the serial–parallel architecture and the serial–serial architecture we have assumed that each node in the computation graph contains one word of delay. This assumption has two important consequences on our implementation: (1) the representation of each PE in memory requires *two* words, one for its previous (last sample-time) output and one for its current output, and (2) the computation for each PE, within one sample-time, is completely independent of the other computations. The second consequence is the reason that architectures employing one processor per node in the computation graph are feasible. Otherwise, the processors could not work in parallel independent of one another. It can also

be used to our advantage in this architecture. Unit delay models have long been used by programs and other *serial* systems that simulate the execution of *parallel* systems. In such systems, one memory is used to "freeze" the state of the system at the end of the previous iteration, while the new state is computed and stored in a second memory. The frozen memory, say A, is used as inputs while the B memory is used as output. After the entire new state is computed, B is used as input and A as output. The process continues, swapping the roles of A and B after each iteration. This technique provides the perfect opportunity to increase the memory bandwidth in our architecture. We replace the memory that holds the outputs of PEs by two memory banks, A and B. This replacement enables independent access to multiplier inputs and outputs, because when A is providing inputs, B in accepting outputs, and vice versa.

Pipelining. Adoption of the unit delay model makes the execution of each PE independent of the others within one sample-time. The input to any multiply–add step will never come from the output of the previous one, nor will it come from any one within its sample-time. This independence allows pipelining to be used to decrease the cycle-time for one operation. The price paid is a decrease in the number of PEs that can be emulated each sample-time. The decrease is equal to the number of sections in the pipeline and is relatively small compared to the total number of PEs being emulated each sample-time. The obvious places to include pipelining stages in the system are between the control memory output and the address decode for the other memory, in the transmission of data to and from the multiplier–adder, and within the multiplier–adder structure.

Array Multiplier. Parallel–parallel multipliers, or *array multipliers*, have been widely studied as to the theoretical bounds for multiplication and efficient implementations. The basic structure of a simple array multiplier is shown in [GLASSER 85]. The structure uses n^2 full-adder cells arranged in a two-dimensional array, each row forming a carry-save adder, and a final row composed of a carry-propagate or some other fast adder circuit. The partial products of the results are summed iteratively in space; therefore, the delay through the structure grows linearly with the number of bits in the words. However, pipelining can be used to reduce the cycle-time to an acceptable level, particularly in our application where latency is not as important as cycle-time. Besides pipelining, two obvious improvements on the basic array structure exist. At the expense of a larger full-adder cell, a modified Booth encoding of the operands can halve the number of rows in the array and thus reduce the delay through the structure by a factor of two. In another modification, the iterative structure composed of the carry-save adders can be changed to a tree structure, commonly called a *Wallace-tree*, which reduces the delay to $O(log(n))$. In a VLSI implementation, such a structure requires extra area for routing wires. Without careful study, it is difficult to predict what the final configuration of the multiplier should be. For the purposes of this study we simply assume that some combination of the above techniques yields a structure with the correct combination of layout area and cycle-time.

An Optimized Architecture. Several of the notions presented above for increasing system performance have been combined in an architecture depicted in Figure 5.6. The basic operation of the system is the same as for the simple system shown in Figure 5.5, with some important improvements. The sound synthesis application requires that the coefficients supplied from the host computer be double buffered. In this implementation we use two separate coefficient memory banks, C_A and C_B. The host can write into one memory bank while the other is used in the computations. When all the desired coefficients are updated, a global bit is set to change to the other coefficient memory. Two memory banks are included, A and B, to implement the unit delay model. On each cycle the control memory provides the address of the inputs and outputs to the multiply–add unit by simultaneously supplying the address for a, b, m, and y. In addition to addresses, the control memory holds the information needed to route the address to the proper memory bank, the data from the memory banks to the proper multiply–add register, and the output to the proper memory bank. Also, the control memory contains information for input–output operation. The non-address information from the control memory is used to set switches that are the control for routing the addresses and data.

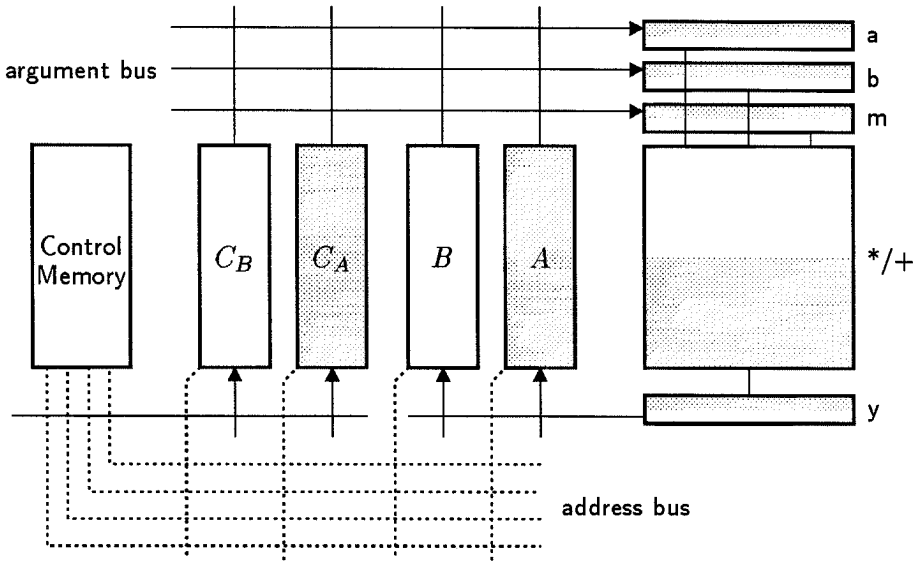

Figure 5.6 Optimized architecture. Parallel memory is provided for reading, writing and coefficients.

The system has been designed so that the operation of the multiply–add unit is in parallel with fetching operands from memory and storing results. Figure 5.7 shows the three possible situations for any PE update cycle and illustrates what activities occur in parallel. Case (a) is the most typical case, in which one PE input is a coefficient and the other inputs are outputs from other PEs. Each row in the chart illustrates the activity of one of the memory banks or the arithmetic unit. In case (a), on the first half of the cycle, the coefficient and first argument for PE_i is

fetched, while the arithmetic operation for PE_{i-1} begins and the result for PE_{i-2} is stored. On the second half of the cycle, the arithmetic operations completes while the second argument is fetched. Case (b) shows the situation when two inputs are coefficients. Most cases are covered by (a) and (b) but occasionally a PE receives no coefficients as inputs. This case is shown in (c) and requires a cycle-time of $1.5\,\tau$, where τ is the minimum cycle time.

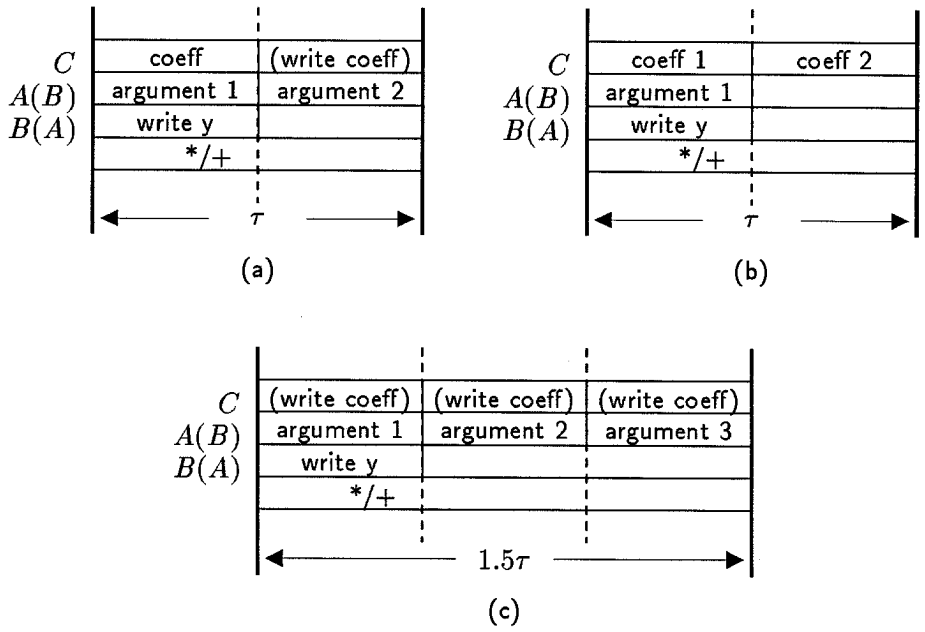

Figure 5.7 Timing diagrams for unit-delay memory arrangement. Separate memories for reading, writing, and coefficients allow simultaneous reads of multiplier inputs and storing of results.

VLSI Implementation

In this section we fill in the details of the architecture presented in Figure 5.6. The system was carefully designed to match the bandwidth of the memory and arithmetic unit in an attempt to maximize the number of PEs that can be emulated per sample-time within the allotted area. We make the same assumptions about technology as for the implementation of the serial–serial architecture in Section *A Serial–Serial Architecture*. All memory is static, $\lambda = 0.8$, yielding a usable chip area of $11,000\,\lambda$ square.

The first step is to estimate the total number of PEs per chip. Based on simulation using the six transistor RAM cell, we assume a memory access time of 20 ns. Therefore, the best-case cycle-time and the value of τ in Figure 5.7 is 40 ns. If we assume that every cycle takes 40 ns, then in one sample-time of $20\,\mu$sec, 500 PE computations are possible. Because some cycles are $1.5\,\tau$ seconds instead of τ, 500

is overly optimistic. Let's assume that one-half of the PEs have inputs from one coefficient, leaving one-quarter with two coefficients and one-quarter with none, because the average is one coefficient per PE. Therefore, one-quarter of the cycles are 1.5τ seconds long and the remainder are τ seconds long. It turns out that the total number of PEs computable in one sample-time is 0.89 times as many as if all cycles were τ seconds long, or in this case about 440. Now the question is: Is there enough chip area to provide memory for 440 PEs? Each PE requires four 32 bit words of memory, plus 40 bits of control for a total of 73,920 bits of memory. Based on preliminary layouts, the full-adder cell is approximately $100 \times 100\,\lambda$, creating about $5000 \times 5000\,\lambda$ for the array multiplier. Based on the layout for RAM cell the memory requires 81.3×10^6 square λ. Assuming we make efficient use of the available memory, 14.7×10^6 square λ remain for memory decoders, address and data buses and various buffers and drivers.

The diagram for the system shown in Figure 5.6 serves well as a guide to the floor plan for the layout. The available vertical dimension in our target area is not sufficient to enable the memory banks to be 440 words deep. Therefore, the memory banks would have to be laid out as 64-bit wide, 220 word deep memory with a multiplexor on the output to select the correct set of bits. The buses and their associated switches are a potential source of large area consumption and require careful examination. Assuming a metal pitch of $7\,\lambda$, the vertical dimension consumed by each 32-bit bus, including switch logic, is $280\,\lambda$ for a total of $840\,\lambda$ for the argument bus. The lower bus requires about $760\,\lambda$, so the buses take up about $1600\,\lambda$ of the $11000\,\lambda$ allotted vertical dimension, leaving sufficient space for the memory arrays.

Power Consumption. We use the same set of assumptions as the memory in the serial–serial design. In this design, the memories are double width with multiplexors on their outputs, but only one-half of the bit lines need to be driven or precharged in every cycle. The control memory is read once each cycle and is 40 bits wide. Also, within one cycle three 32-bit operands are read and one 32-bit result is written. Following is a summary of the capacitance calculations:

control memory read	97.9 pF
three 32-bit operand reads	235
one 32-bit write	39.2
buses	23.0
multiply–adder	45.6
total capacitance	441 pF.

Therefore, the total chip energy per clock cycle is

$$E_{chip} = 11.0 \times 10^{-9}\,\text{Joules},$$

and the power consumption at the design speed of 25 MHz is

$$P = 0.276\,\text{Watts}.$$

In this case,

$$E_{OP} = E_{slice}.$$

Other Improvements

In the design presented above we have not used all the techniques available to increase the memory bandwidth, nor have we pushed the limits of the attainable cycle-time for the array multiplier.

One obvious extension to our architecture is to supply one A–B memory bank pair for each of `a`, `b`, and `m` inputs. This extension reduces the typical cycle-time to one memory access. However, outputs from PE's that fanout to more than one type of input would have to be represented in more than one memory bank. This redundancy requires that some results be written to more than one memory and also a larger overall memory size.

Perhaps all the techniques can be used simultaneously. This particular arrangement was chosen because it provides a good balance between the size and speed of the components. Any attempts to speed up the system, however, invariably increase its size. The reason for this tradeoff is two-fold: (1) the components gain speed by trading off size; for instance, dual-porting the memory increases its speed but also its area; and (2) a shorter cycle means that more PEs can be emulated during each sample-time, and therefore more memory is needed to hold their state and coefficients.

It appears that within the technological limits we have assumed that the only way to exploit a shorter cycle-time is to use multiple chips for each parallel–parallel machine. Severe speed penalties are paid for crossing chip boundaries. This problem is commonly solved by using a local cache memory on the same chip as the arithmetic unit and a larger memory bank off-chip. With today's technology about a factor of four speedup in cycle-time is possible, but the statistics of the cache behavior heavily influences overall system performance.

Throughout this section on the parallel–parallel architecture we have assumed that each PE includes one word of delay. This assumption enabled us to treat PEs independently within a sample-time but also increased the amount of memory required. In reality, many PEs do not require delay and algorithms could be revised to exploit a zero-delay node. For those PEs requiring zero delay, the memory requirement reduce because registers are required only to hold temporary results and can be reused within the same cycle-time. Also no state registers are required. This approach may increase the cycle-time, however, because one PE computation may have to wait for the results from a previous one, destroying the benefits of the pipeline. This increase in cycle-time can be minimized if a sufficient mix of unrelated computations is present to keep the pipeline full and to exploit dual memory banks. Any practical architecture would have capabilities of supporting both types of nodes. I have focused on the delay-per-node model because it corresponds directly to the model supported by the other architectures.

Serial–Parallel Architecture Summary

Based on the circuits from Chapter 4, *VLSI Implementation*, and their associated layout sizes, we estimate the area, speed, and power consumption for the architecture based on the serial–parallel processor.

Layout Area. The details of this design are presented in Chapter 4, *A VLSI Architecture*, and the floor plan is shown in Figure 5.2 of that chapter. As with the serial–serial approach, the estimates can be made for a single *slice* comprising a processing element with its internal registers and a section of the connection matrix. In this design, unlike the serial–serial design, all the coefficient registers are grouped together in one block, and channels are provided in the switch to distribute the coefficient data to the PEs. The block contains one double-buffered coefficient register per PE. In the calculation we group a single coefficient register with each PE for accounting purposes.

Assuming a horizontal orientation for the PEs, each PE is 3340 wide by 208 high. We add 52 in the vertical dimension for a coefficient register for a total of $260\,\lambda$. Assuming $11000\,\lambda$ square as the usable chip area, we estimate the total number of PEs in a column at 40. As with the serial–serial design, we include 10 channels of switch for inter-PE communication. In this case we must also allow 40 extra channels for coefficient-to-PE communications. The horizontal dimension contributed by the switch to the slice is 1950, for a total slice width of 5300. One column of PE slices fills the chip area in the vertical dimension and approximately one-half of the horizontal dimension. Two columns of PEs fit, with little waste, for a total of 80 PEs.

Speed. For sound-synthesis applications with a sample rate of 50 KHz the bit rate for this system is $50\,\text{KHz} \times 64\,\text{cycles/sample} = 3.2\,\text{MHz}$. Other applications may require a higher sample rate. If we allow the system to run with the bit rate of the serial–serial architecture, 54.4 MHz, a sample rate of 850 KHz is achieved. The total computation rate for the chip is 68 million operations per second (MOPS).

Power. As with the other designs, we estimate the power for this architecture based on the cell layouts and floor plan for one slice. The calculations are summarized below:

$$\begin{aligned} \text{switch data lines} &: 1.91\,\text{pF} \\ \text{coefficient buffer bit lines} &: 0.523 \\ \text{PE shift registers} &: 2.56 \\ \text{PE nodes} &: 3.57 \\ \text{total capacitance} &: 8.56\,\text{pF}. \end{aligned}$$

Therefore, the energy per slice is

$$E_{slice} = 214 \times 10^{-12}\,\text{Joules};$$

at $V_{dd} = 5.0\,\text{volts}$, and the total chip energy per clock cycle is

$$E_{chip} = 17.1 \times 10^{-9}\,\text{Joules}.$$

Table 5.2 Normalized comparison.

Architecture	Area Per PE ($1000 \times \lambda^2$)	PE Rate (KOPS)	Energy Per Operation (nj/OP)*	Computations Per Area (OPS/λ^2)
serial–serial	315	50	67.8	0.159
serial–parallel audio rate	1513	50	13.7	0.033
serial–parallel full speed	1513	850	13.7	0.562
parallel–parallel	275	50	11.0	0.182

* power–delay product

The energy associated with one slice is used $2n$ times per operation; therefore,

$$E_{OP} = 2n \cdot E_{slice} = 13.7\,\text{nJ}.$$

At the clock rate for sound synthesis and audio processing of 3.2 MHz, the power consumption is

$$P_{audio} = 54.7\,\text{mW}.$$

At the full-speed clock rate of 54.4 MHz, the total power consumption of the chip is

$$P_{full\text{-}speed} = 0.930\,\text{Watts}.$$

Comparing the Architectures

The results of the estimates made in the preceding three sections are shown in Tables 5.1 and 5.2. Four sets of figures are presented in each table—one for the serial–serial and parallel–parallel architectures and two for the serial-parallel architecture, one for audio sampling rate and the other for maximum sampling rate, or full-speed operation. Table 5.1 compares the architectures on a per-chip basis, showing figures that apply to the entire chip. In Table 5.2 similar figures are normalized to a per-PE or per-operation basis.

Table 5.1 Comparison per chip.

Architecture	Number of PEs	Total Computational Rate (MOPS)	Power (watts)
serial–serial	384	19.2	1.30
serial–parallel audio rate	80	4.0	0.055
serial–parallel full speed	80	68.0	0.930
parallel–parallel	440	22.0	0.276

It is important not to interpet these figures as being accurate in general for each of the three classes of arithmetic; they are specific to the particular architectures chosen and include not only the arithmetic units but also coefficient buffers and the interconnection mechanism. The figures do represent, however, a good estimate of results that are attainable with current technology for this and for other similar tasks.

The serial–parallel architecture is represented by two rows of numbers in each table. The first row is at audio rate, 50 K samples/second. This architecture is capable of a much higher sample rate, but a higher sample rate is wasted with sound synthesis. The second row for the serial–parallel architecture applies with its maximum sample rate, as it may for other applications. The fact that the serial–parallel design must run at a slow speed for sound synthesis motivated the other two designs. At audio-rate applications both the serial–serial and the parallel–parallel architectures implement approximately five times the number of PEs per chip as the serial–parallel approach.

The serial–serial approach trades speed away for area. The number of full-adder cells scales down by a factor equal to n, the number of bits in the word, but the connection matrix circuits and the register associated with inputs, outputs, and temporary results do not. The result is that the area of a PE is only five times smaller for the serial–serial design than for the serial–parallel one. The serial–serial approach, because of its n^2 memory accesses per operation has a high power consumption.

The parallel–parallel approach trades area for speed and time multiplexes its arithmetic unit. Its power budget is comparable with the serial–parallel approach as is evident from its energy-per-operation figure.

Comparing the architectures for pure computation power, the serial–parallel approach running at full speed performs the most operations/second/area, and the entire chip achieves a total computational rate of 68 million operations per second.

Conclusion

This thesis presents (1) a new approach to the production of musical sounds; (2) the application of this approach to several musical instrument models; (3) the design of a custom computing engine to support the approach; (4) the details of a VLSI implementation of the computing engine; and (5) the presentation of several related architectures.

Our solution to the problem of sound synthesis is one that employs the flexibility provided by VLSI to build an architecture that is tailored to the computation involved in modeling the dynamics of musical instruments. The key to the efficiency of our machine differentiates it from other concurrent architectures; no processing cycles are used for communication; the processors are dedicated to arithmetic operations and the connection strategy is preprogrammed to provide the communication for a specific task.

The architectures presented exploit the natural parallelism of the problem at every possible level. At the lowest, or *logic*, level pipelining is used to efficiently implement the arithmetic operations. At the processor, or *arithmetic level*, processing elements work independently to implement the instrument models. Within the instrument models at the *difference equation level*, various pieces of the models are computed concurrently. At the highest level, or *voice level*, instruments of an ensemble or voices of a multivoiced instrument are computed simultaneously.

This thesis described two simple musical instrument models. With current integrated circuit technology it is possible to realize, per chip, approximately 10 instruments of this type. Therefore, a moderate-sized system is capable of producing the sound of hundreds of voices.

Although the instrument models have been used to produce extremely high-quality timbres of certain instruments, they are certainly not capable of covering the entire range of timbres of the instruments. They are simplistic models of the physics of the musical instruments that they emulate, and are meant as examples and a basis for future study. The activity of modeling requires a great deal of careful study and will involve extensions and modifications to the models.

Results in modeling may lead to architectural changes. A possible change in the architecture may be to incorporate into its design the use of large amounts of temporary memory configured as delay lines of the type required for *scattering models.* Commercially available memory could be used in this application. Scattering models are efficient for emulating wave propagation in uniform mediums such as uniform air columns and for simulating physically large systems such as reverberant performance halls. The approach we have presented is more general but may be relatively expensive in some simple cases. A hybrid system could include the best of each approach.

Musical sound synthesis has many attributes in common with other problems in science and engineering. The computations required for modeling musical instruments are representative of a larger class of problems that can be formulated using systems of finite difference equations. Our architecture should be equally efficient for these related problems. These are problems where a fixed (or slowly varying) interconnection of elements is sufficient. Once the interconnection topology is defined, the computation proceeds for a relatively long period before another interconnection change is made. Conventional signal processing can be viewed in this manner. In general, this class of problems comprises those that may be represented as systems of difference equations, where *time in the problem* being modeled may be represented by *time in the computation.* Our belief is that the architecture presented here will find general application among this class of problems, as an efficient and sometimes necessary alternative to general purpose computers.

Appendix

Digital Resonators

This appendix further develops the basic theory of second-order digital resonators. Section *Basic Equations* reiterates the basic equations governing the behavior of two-pole sections. Section *Q Calculation* contains a discussion of Q calculation, as a user parameter. Section *Resonance Gain* presents the problems with resonator gain and a proposed solution. Finally, the special case of critically damped sections is presented.

Basic Equations

Above critical damping, the system function for a digital resonator can be formulated with the finite-difference equation:

$$H(z) = \frac{1}{(1 - Re^{j\theta_c}z^{-1})(1 - Re^{-j\theta_c}z^{-1})}. \tag{1}$$

The UPE implementation of a digital resonator contains the additional term in the numerator, z^{-2}, representing a pure delay of two word times. This term may be factored out and will not be treated here. Multipling out the denominator, we get

$$H(z) = \frac{1}{1 - 2R\cos\theta_c z^{-1} + R^2 z^{-2}}, \tag{2}$$

and the difference equation

$$y_n = 2R\cos\theta_c y_{n-1} - R^2 y_{n-2} + x_n. \tag{3}$$

Equation 2 leads to a sinusoidal time-domain impulse response of the form

$$\gamma R^n \sin(n\theta_c + \phi), \tag{4}$$

where $\gamma = 1/\sin\theta_c$ and $\phi = \theta_c$ from the partial fraction expansion of Equation 2. For values of $R < 1$, the response is a damped sine wave with R controlling the rate of damping and θ_c controlling the frequency of oscillation.

Q Calculation

Some confusion exists as to what Q actually means or should mean for digital resonators. Traditionally, Q has been a measure of the rate at which a system loses energy. In other words, Q is the time, measured in cycles, that the system takes to reach some percentage of its original value. In the frequency domain, Q measures the bandwidth of the pass-band. However one defines Q, the user should be able to use Q to control the rate of damping of the resonator or the width of the pass-band in the frequency domain.

When we view the digital resonator as a time-domain device, it may be that the simplest way for the user to control the damping rate is to control R directly in Equation 3. However, R is not in convenient units, and for musical applications is nearly always very close to one. This problem leads to an approximation for Q that is simple to compute and provides the user with a convenient control of damping rate. Let Q be *the number of cycles until the impulse response of the system reaches* $1/e$ *of its original value.*

To find R in terms of Q, first we find how many samples until the system reaches $1/e$ of its orginal value. Using the factor R^n from Equation 4:

$$R^n = 1/e \quad \Rightarrow \quad n = \frac{-1}{\ln R} \text{ samples.}$$

To convert from samples to cycles, we multiply by f/f_s, where f is the resonator center frequency and f_s is the sampling frequency:

$$Q = -\frac{f}{f_s \ln R},$$

so

$$R = e^{-f/f_s Q}.$$

For values of R close to unity, $-1/\ln R \approx 1/(1-R)$; therefore, we can approximate R by

$$R = 1 - \frac{f}{f_s Q}.$$

I present another interpretation of Q having the property that, at $Q = 0.5$, the system is critically damped. Q is defined as it is for a continuous system. We can view the time-domain response of the digital resonator as a sampled version of the impulse response of a continuous resonator. The continuous resonator can be analyzed, using the placement of its poles in the s-plane. For a resonator with complex conjugate poles and a damped response, the poles lie at $s = -\sigma \pm j\omega_{CF}$, that is, to the left of the imaginary axis at a distance σ, and above and below the real axis a distance ω_{CF} (center frequency), as illustrated in Figure A.1. The radial distance from the origin to a pole is called ω_N (natural frequency). The distance σ controls the damping rate.

We define Q to be $\omega_n/2\sigma$. Note that the center frequency ω_{CF} is a function of Q. From a geometric argument,

$$\omega_{CF} = \omega_N \sqrt{1 - \left(\frac{1}{2Q}\right)^2}, \quad \text{for} \quad Q \geq 0.5. \tag{5}$$

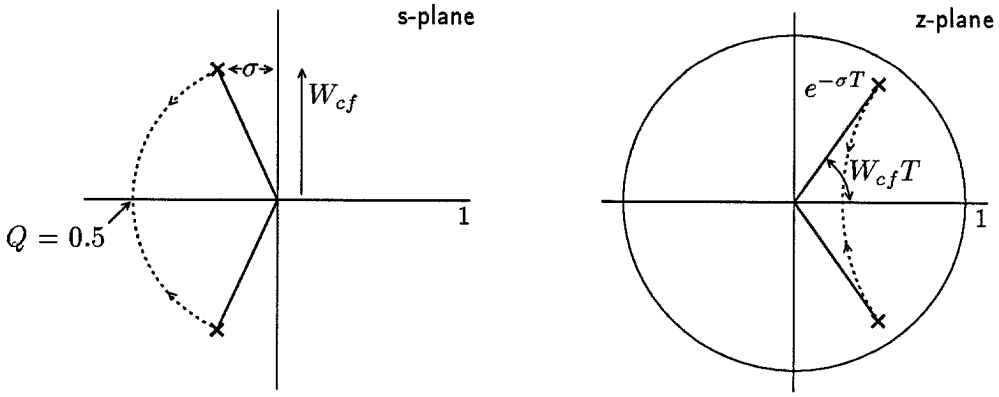

Figure A.1 Pole placements for continuous and digital resonators.

Converting one pole to the z-plane:

$$z = e^{sT} = e^{(-\sigma + j\omega_{CF})T} = e^{-\sigma T} e^{j\omega_{CF} T}.$$

Now, given a Q, and a frequency, θ_c and R are computed by:

$$\begin{aligned} R &= e^{-\sigma T} \\ \sigma &= \frac{\omega_N}{2Q} \\ \theta_c &= \omega_{CF} T. \end{aligned}$$

Note in Equation 5 that when $Q = 0.5, \omega_{CF}$ goes to zero. Here there is a double root on the real axis—the system is critically damped.

Resonance Gain

One important property of digital resonators is their gain at resonance. This gain can be found by taking the magnitude of the discrete Fourier transform evaluated at the resonant frequency. Substituting $e^{j\theta_c}$ for z in Equation 1, we obtain

$$\begin{aligned} H(e^{j\theta_c}) &= \frac{1}{(1 - Re^{j\theta_c} e^{-j\theta_c})(1 - Re^{-j\theta_c} e^{-j\theta_c})} \\ &= \frac{1}{(1 - R)(1 - Re^{-2j\theta_c})}. \end{aligned} \tag{6}$$

$$Gain = |H(e^{j\theta_c})| = \frac{1}{1 - R} \cdot \frac{1}{\sqrt{1 - 2R\cos(2\theta_c) + R^2}}.$$

Figure A.2 shows the gain at resonance plotted for various values of R from frequency zero to $f_s/2$.

The gain at resonance varies drastically over the frequency range. This variation causes scaling problems when fixed-point arithmetic is used. Either the input to or the output from each resonator must be adjusted to compensate for the implicit gain of the resonator. Several techniques exist for normalizing resonator gain.

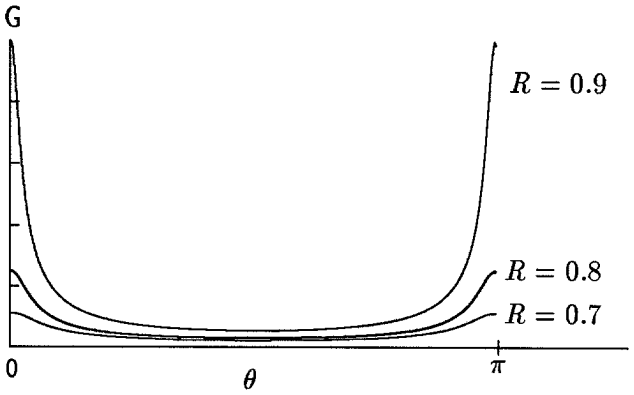

Figure A.2 Resonator gain. Shown at resonance as a function of center frequency, plotted for several values of R.

One proposed by Smith and Angell, uses the addition of two zeros to the second-order system function [SMITH 82]. By placing a zero at $\pm\sqrt{R}$, we can eliminate the dependence on θ in the system function. The new system function uses Equation 1 and multiplies it by $1 - Rz^{-2}$. Evaluating at $z = e^{j\theta_c}$ yields

$$\begin{aligned} |H(e^{j\theta_c})| &= \frac{1 - R(e^{-j\theta_c})^2}{(1 - Re^{j\theta_c}e^{-\theta_c})(1 - Re^{-j\theta_c}e^{-\theta_c})} \\ &= \frac{1 - Re^{-2j\theta_c}}{(1 - R)(1 - Re^{-2j\theta_c})} \\ Gain &= \frac{1}{1 - R}. \end{aligned} \tag{7}$$

Resonator-gain normalization could pose a particularly severe problem in the case of resonator banks of the type shown in Figure 2.14, an important structure in sound synthesis. Scaling the input to each resonator increases the amount of hardware by a factor of approximately one-third and increases the control bandwidth by the same amount. Alternatively, the input to the entire system can be scaled down, to avoid overflow in the section with the most gain, and then the output scaled up to the appropriate level. This approach is a problem in systems that use fixed-point arithmetic; usually, the largest coefficient possible is relatively close to one. Therefore, not much gain is possible, and many gain stages at the output must be used.

In many sound-generation applications, the R values of each stage in the resonator bank are close to one another. Therefore, it is possible to synthesize two zeros using an average or some other value for R and then distributing the result to each resonator, as shown in Figure A.3. The input $x(n)$ passes through a section that generates two zeros before being distributed to the resonators.

Resonance Phase

Phase is important when resonators are included in feedback loops. As is true of the gain, the phase at resonance of Equation 1 is not constant. The phase at resonance

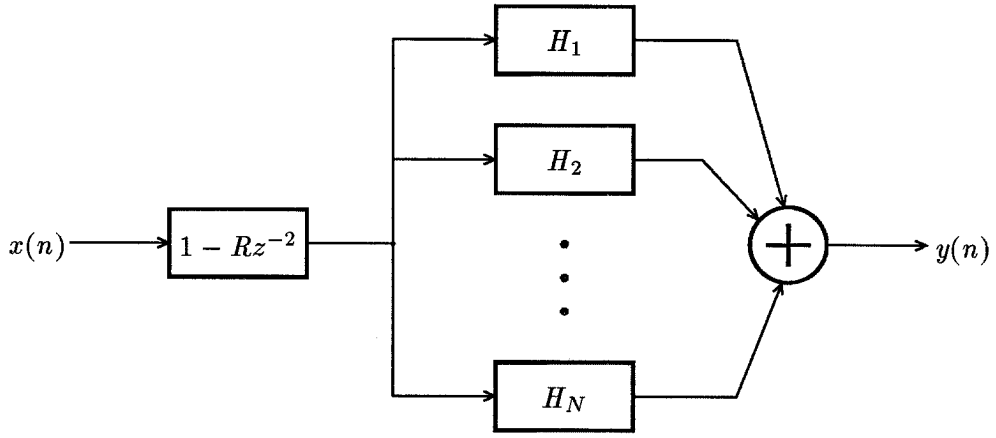

Figure A.3 Normalized resonator bank. Two zeros are synthesised and distributed to all the resonators.

is found by computing the angle of Equation 6:

$$\begin{aligned}\Phi\left[H(e^{j\theta_c})\right] &= \Phi\left[\frac{1}{(1-R)(1-Re^{-2j\theta_c})}\right] \\ &= \Phi\left[\frac{1}{1-R\cos(2\theta_c)-jR\sin(2\theta_c)}\right] \\ &= \tan^{-1}\left[\frac{R\sin(2\theta_c)}{1-R\cos(2\theta_c)}\right].\end{aligned}$$

The phase at resonance is shown for several values of R in Figure A.4.

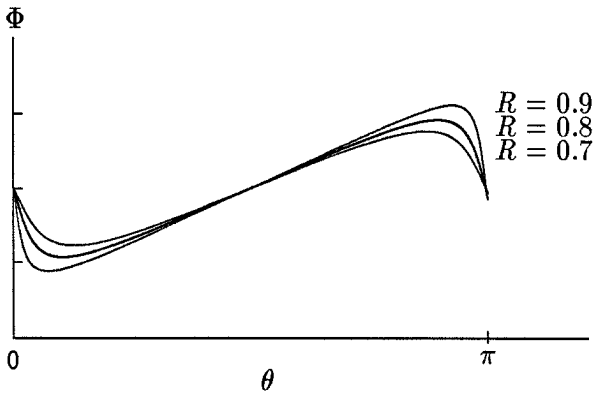

Figure A.4 Phase at resonance. Plotted as a function of center frequency for several values of R.

It is interesting to examine the effect of the addition of two zeros to the second-order system function by placing a zero at $\pm\sqrt{R}$. From Equation 7, it is clear that the normalized resonator has zero phase response at resonance for all θ_c and R.

Time-Domain Response of Critically Damped Resonator

An important special case of Equation 1 is when $\theta_c = 0$ and a double pole exists on the real axis. Such a system is said to be *critically damped* and has an impulse response as shown in Figure A.5.

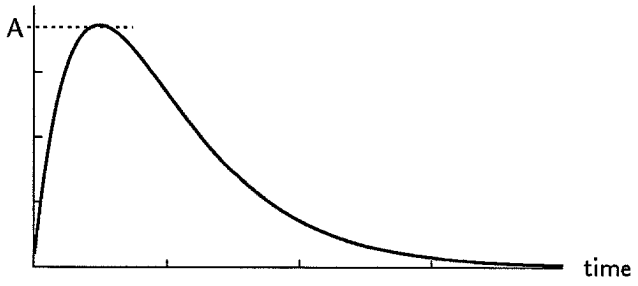

Figure A.5 Impulse response of critically damped resonator.

Following the Q calculation based on continuous resonators in Section *Q Calculation*, critical damping occurs at $Q = 0.5$. Systems with Q values close to or equal to 0.5 can be used as driving functions to resonator banks for certain simple struck-instrument models.

The placement of the double pole, and thus of τ in Figure A.5, is directly controlled by ω_N, because at $Q = 0.5$, $\sigma = \omega_N$ and $R = e^{-\omega_N T}$.

The maximum value of the time-domain response, A in Figure A.5, varies as a function of R. Consider the system function, Equation 1, with $\theta_c = 0$:

$$H(z) = \frac{1}{(1 - Rz^{-1})^2}.$$

The inverse z-transform can be found by using the power-series expansion for $(1 - x)^{-n}$:

$$\frac{1}{(1-x)^n} = \sum_{k=0}^{\infty} \frac{(n+k-1)!}{(n-1)! \cdot k!} x^k.$$

Letting $x = Rz^{-1}$ and $n = 2$ yields

$$H(z) = \sum_{k=0}^{\infty} (k+1) R^k z^{-1}.$$

By the definition of the z-transform,

$$h(k) = (k+1)R^k.$$

To find the maximum value, we find the first derivative of $h(k)$ with respect to k, $h'(k)$, and solve for k when $h'(k) = 0$.

$$h'(k) = [1 + (1 + k)\ln R]R^k$$

$$k_{max} = -\left(1 + \frac{1}{\ln R}\right).$$

Evaluating $h(k)$ at k_{max} yields:

$$h_{max}(k) = -\frac{1}{eR\ln R}.$$

References

[ABELSON 80] Abelson, H. and Andreae, P. (1980). Information Transfer and Area–time Tradeoffs in VLSI Multiplication. *Communications of the ACM.* 32(1): 20–23.

[BRENT 80] Brent, R. P., Kung, H. T. (1980). The Chip Complexity of Binary Arithmetic. *Proc. 12th ACM Symp. Theory Comput.* pp. 190–200.

[BRILLOUIN 46] Brillouin, L. (1946). *Wave Propagation in Periodic Structures.* McGraw-Hill: New York, NY.

[CHEN 83] Chen, M. (1983). *Space–Time Algorithms: Semantics and Methodology.* California Institute of Technology, Computer Science Department Technical Report 5090:TR:83.

[CHENG 76] Cheng, E. K., & Mead C. A. (1976). A Two's Complement Pipeline Multiplier. *Proceedings of 1976 IEEE International Conference on Acoustics, Speech and Signal Processing*, Philadelphia, PA.

[CHOWNING 73] Chowning, J. M. (1973). The Synthesis of Complex Audio Spectra by Means of Frequency Modulation. *Journal Audio Engineering Society* 21(7): 526–534.

[DAS 23] Das, P. (1923). On the Impact of an Elastic Hammer on a Pianoforte String. *Indian Journal Physics* 10: 75–96.

[DENYER 83] Denyer, P. B. (1983). An Introduction to Bit-Serial Architectures for VLSI Signal Processing. In *VLSI Architecture,* editors: B. Randall and P. C. Treleven, Prentice-Hall: Englewood Cliffs, NJ.

[DYER 87] Dyer, L. (1987). Masters Thesis in preparation. Computer Science Department, Caltech.

[FLETCHER 80] Fletcher, N. H. and Douglas, L. M. (1980). Harmonic Generation in Organ Pipes, Recorder and Flutes. *Journal Acoustic Society* 68(3).

[FLETCHER 83] Fletcher, N. H. and Thwaites S. (1983). The Physics of Organ Pipes. *Scientific American* 248(1):84–93.

[FRIEDLANDER 53] Friedlander, F. G. (1953). On the Oscillations of the Bowed String. *Proc. Cambridge Philos. Soc.*, 49: 516–530.

[GLASSER 85] Glasser, L. A. and Dobberpuhl, D. W. (1985). *The Design and Analysis of VLSI Circuits* Addison Wesley: Reading, MA.

[GORDON 85] Gordon, J. W. (1985). System Architectures for Computer Music. *Computing Surveys.* Vol. 17, No. 2, pp. 191–233.

[HELMHOTZ 54] Helmhotz, H. L. F. (1954). *On the Sensations of Tone.* 2nd English Ed., based on 4th German Ed., 1877, A. Ellis, Ed., Dover Publications: New York.

[HILLER 71] Hiller, L. and Ruiz, P. (1971). Synthesizing Musical Sounds by Solving the Wave Equation for Vibrating Objects: Parts I and II. *Journal Audio Engineering Society* 19(6): 462–470 and 19(7): 542–550.

[JACKSON 68] Jackson, L. B., Kaiser, S. F. and McDonald, H. S. (1968). An Approach to the Implementation of Digital Filters, *IEEE Transactions on Audio and Electroacoust.*, AU-16: 413–421.

[JACKSON 69] Jackson, L. B. (1969). An Analysis of Limit Cycles due to Multiplication Rounding in Recursive Digital (Sub) Filters, *Proc. 7th Annu. Allerton Conf. Circuit System Theory*, pp. 69–78.

[JAFFE] Jaffe, D. A. and Smith, J. O. (1983). Extensions of the Karplus-Strong Plucked String Algorithm. *Computer Music Journal.* 7(2): 56–69.

[KARPLUS 83] Karplus, K. and Strong A. (1983). Digital Synthesis of Plucked-String and Drum Timbres. *Computer Music Journal.* 7(2): 43–55.

[KELLER 53] Keller, J. B. (1953). Bowing of Violin Strings. *Applied Mathematics*, 6: 483–495.

[KNUTH 68] Knuth, D. E. (1968). *The Art of Computer Programming, Vol. 2.* Addison Wesley: Reading, MA.

[LUK 81] Luk, W. K. (1981). A Regular Layout for Parallel Multiplier of $O(Log^2N)$ Time. In *VLSI Systems and Computations*, editors: H. T. Kung, Bob Sproull, and Guy Steele, Computer Science Press: Rockville, Maryland, pp. 317–326.

[LYON 76] Lyon, R. F. (1976). Two's Complement Pipeline Multipliers, *IEEE Transactions on Communications*, April 1976, pp. 418–425.

[LYON 81] Lyon, R. F. (1981). A Bit-Serial VLSI Architecture Methodology for Signal Processing. *VLSI 81 Very Large Scale Integration, (Conf. Proc., Edinburgh, Scotland, John P. Gray, editor)* Academic Press: New York, NY.

[LYON 85] Lyon, R. F. (1985). MSSP: A Bit-Serial Multiprocessor for Signal Processing. In *VLSI Signal Processing*, Denyer & Renshaw, Addison Wesley: Reading, MA, pp. 263–275.

[MAGANZA 86] Maganza, C., Causse, R. and Laloë F. (1986). Bifurcations, Period Doublings and Chaos in Clarinetlike Systems. *Europhys. Lett.* 1(6): 295–302.

[MCINTYRE 83] McIntyre, M. E., Schumacher, R. T. and Woodhouse J. (1983). On the Oscillations of Musical Instruments. *Journal Acoustic Society America* 74(5).

[MEAD 80] Mead, C. A., & Conway L. A. (1980). *Introduction to VLSI Systems*, Chapter 9. Addison Wesley: Reading, MA.

[MEAD 85] Mead, C. A. and Wawrzynek J. C. (1985). A New Discipline for CMOS Design: an Architecture for Sound Synthesis, *1985 Chapel Hill Conference on Very Large Scale Integration*, edited by Henry Fuchs, Computer Science Press.

[MOORE 78] Moore, J. L. (1970). Acoustics of Bar Percussion Instruments. Ph.D. Dissertation, Music Department, Ohio State University.

[MOORER 70] Moorer, J. A. (1977). Signal Processing Aspects of Computer Music: A Survey. *Proceedings of the IEEE.* 65(8): 1108–1137.

[MORSE 36] Morse, P. M. (1936). *Vibration and Sound.* McGraw-Hill: New York, NY.

[MOSIS USER'S MANUAL 86] MOSIS — MOS Implementation System (1986). *User's Manual.* USC information Sciences Institute, Marina Del Rey, CA.

[OPPENHEIM 75] Oppenheim, A. V. and Schafer, R. (1975). *Digital Signal Processing.* Prentice-Hall: Englewood Cliffs, New Jersey.

[PICKLES 82] Pickles J. O. (1982). *An Introduction to the Physiology of Hearing*, Academic Press: Orlanda, Florida.

[PIERCE 86] Pierce, J. R. (1986). *Comments on Analysis and Synthesis of Musical Sounds.* Unpublished Report, Center for Computer Research in Music and Acoustics (CCRMA), Dept. of Music, Stanford University, Stanford, CA.

[PRINCE 83] Prince, B., and Due-Gundersen, G. (1983). *Semiconductor Memories*, John Wiley & Sons: New York.

[RADAR 67] Radar, C. M. and Gold B. (1967). Digital Filter Design Techniques in the Frequency Domain. *Proceedings of the IEEE.* 55: 149–171.

[RAYLEIGH 45] Rayleigh, J. W. S. (1945). *The Theory of Sound.* Dover Publications: New York, NY.

[ROADS 85] Roads, C. and Strawn J. (1985). *Foundations of Computer Music.* MIT Press, Cambridge, MA.

[ROSSING 82] Rossing, T. D. (1982). Acoustics of Bar Percussion Instruments. *Percussive Notes* 19(3): 6–17.

[SEITZ 84] Seitz, C. L. (1984). Concurrent VLSI Architectures. *IEEE Transactions on Computers* C-33(12): 1247–1265.

[SMITH 82] Smith, J. O. and Angell J. (1982). A Constant-Gain Digital Resonator Tuned by a Single Coefficient. *Computer Music Journal* 6(4).

[SMITH 85a] Smith, J. O. (1985a). *Waveguide Digital Filters.* Internal Report, Center for Computer Research in Music and Acoustics (CCRMA), Dept. of Music, Stanford University, Stanford, CA.

[SMITH 85b] Smith, J. O. (1985b). A New Approach to Digital Reverberation using Closed Waveguide Networks. *Proceedings 1985 International Conference Computer Music*, Vancouver, Canada. Computer Music Association, also published as, Music Dept Tech. Rep. STAN-M-31, Stanford University, Stanford, CA.

[SMITH 86] Smith, J. O. (1986). Efficient Simulation of the Reed-Bore and Bow-String Mechanisms. *Proceedings 1986 International Conference Computer Music, The Hague, The Netherlands.* Computer Music Association.

[THOMPSON 80] Thompson, C. D. (1980). *A complexity theory for VLSI.* Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Technical Report CMU-CS-80-140.

[WAWRZYNEK 84] Wawrzynek, J. C. and Tzu-Mu Lin (1984). A Bit Serial Architecture for Multiplication and Interpolation. California Institute of Technology, Computer Science Department, Pasadena, CA. Display File 5067.

[WAWRZYNEK 85a] Wawrzynek, J. C., & Mead C. A. (1985a). A VLSI Architecture for Sound Synthesis. In *VLSI Signal Processing*, Denyer & Renshaw, Addison Wesley: Reading, MA, pp. 277–297.

[WAWRZYNEK 85b] Wawrzynek, J. C., Tzu-Mu Lin, Mead C. A., Liu H., & Dyer L. (1985b). *A VLSI Approach to Sound Synthesis.* (Conf. Proc., 1984 International Computer Music Conference, William Buxton, editor), Paris, France.

[WEINREICH 79] Weinreich, G. (1979). The Coupled Motions of Piano Strings. *Scientific American* 240(1): 118–127.